SEPARATING PROTECTION AND MANAGEMENT IN CLOUD INFRASTRUCTURES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Zhiming Shen

December 2017

SEPARATING PROTECTION AND MANAGEMENT IN CLOUD INFRASTRUCTURES

Zhiming Shen, Ph.D.

Cornell University 2017

Cloud computing infrastructures serving mutually untrusted users provide security isolation to protect user computation and resources. Additionally, clouds should also support flexibility and efficiency, so that users can customize resource management policies and optimize performance and resource utilization. However, flexibility and efficiency are typically limited due to security requirements. This dissertation investigates the question of how to offer flexibility and efficiency as well as strong security in cloud infrastructures.

Specifically, this dissertation addresses two important platforms in cloud infrastructures: the containers and the Infrastructure as a Service (IaaS) platforms. The containers platform supports efficient container provisioning and executing, but does not provide sufficient security and flexibility. Different containers share an operating system kernel which has a large attack surface, and kernel customization is generally not allowed. The IaaS platform supports secure sharing of cloud resources among mutually untrusted users, but does not provide sufficient flexibility and efficiency. Many powerful management primitives enabled by the underlying virtualization platform are hidden from users, such as live virtual machine migration and consolidation.

The main contribution of this dissertation is the proposal of an approach inspired by the exokernel architecture that can be generalized to any multi-tenant system to improve security, flexibility, and efficiency. This approach is called

the exokernel approach — a principle of separating protection and management. By separating protection and management, the protection layer can focus on security isolation and resource multiplexing, making security guarantees easier to maintain and verify. Resource management components are dedicated to each user or application for customization and optimization, greatly improving flexibility and efficiency. We investigate the effectiveness of this approach by applying it to the containers and the Infrastructure as a Service (IaaS) platforms, and introduce X-Containers and Library Cloud. X-Containers is a new exokernel+LibOS architecture that is fully compatible with Linux containers and provides competitive or superior performance to native Docker Containers as well as other LibOS designs. Library Cloud is a new abstraction that enables more flexible and efficient user-level cloud resource management without breaking security isolation between different users. Together, these systems represent important steps towards secure, flexible, and efficient cloud infrastructures.

BIOGRAPHICAL SKETCH

Zhiming Shen grew up in Fujian, China. Although both of his parents are Chinese language teachers, he has an innate enthusiasm for computer architecture and programming. He attended Shantou University from 2001 to 2005, majoring in Applied Mathematics. In 2006 he attended Peking University and in 2010 obtained a Master of Software Engineering degree. During this period, Zhiming had the opportunity to work as a full-time intern for both Microsoft Research Asia and Intel Labs China. These experiences greatly expanded his personal vision and informed his decision to pursue a Ph.D. in Computer Science.

Zhiming's Ph.D. journey started in 2010 at North Carolina State University, where he studied cloud resource management. The research experience at NCSU piqued his interest in virtualization technology and cloud infrastructures, which motivated him to look for a program where he could further explore fundamental research questions. In 2013, he restarted his Ph.D. adventure at Cornell University. He enjoyed his time at Cornell immensely, relishing the opportunity to work with his two wonderful advisers researching topics that truly excited him. He successfully defended his Ph.D. dissertation in November, 2017.

Although it was a long journey to his current position at Cornell, full of unexpected twists and turns, Zhiming is grateful for his experiences, even the difficulties he was forced to overcome, all of which made him a stronger researcher and person. He and his family now look forward to starting a new chapter in their lives.

Success is not final. Failure is not fatal. It is the courage to continue that counts.

For my wife and parents

who always support me, stand by me, and pray for me.

ACKNOWLEDGEMENTS

First, I would like to express my deepest gratitude to my advisers, Robbert van Renesse and Hakim Weatherspoon, who guided me during my graduate studies and helped me become an independent researcher. Robbert is always inspiring and a great source of help. From him I learned a lot about how to formulate ideas, conduct research, and present convincing results in papers. His wisdom on system research is always amazing, and greatly widened my horizons. Hakim is an endless source of warmth, support, patient guidance, and optimistic spirit. He gave me the courage to pursue fundamental research questions, which had a marked influence on my work attitude. I am also grateful to Andrew Myers and Shawn Mankad, who graciously agreed to become my thesis committee members. Their advice and comments regarding my dissertation have proven invaluable.

I was fortunate enough to work with many excellent research collaborators at Cornell, without whom this dissertation would not be possible. I remember the time I spent working with Qin Jia, Weijia Song, Gur-Eyal Sela, Eugene Bagdasaryan, and Ben Rainero with great fondness, cheering for good experiment results, struggling with performance bugs, writing papers until late at night, and celebrating paper acceptance. I am grateful for the opportunity to work with Christina Delimitrou, who gave me a lot of invaluable feedback on my research, and helped me to greatly improve my presentations.

The Department of Computer Science at Cornell is a warm community. I would like to thank faculty members, especially Lorenzo Alvisi, John Hopcroft, Kavita Bala, and Rachit Agarwal, for all their helpful advice. I would also like to thank my colleagues, who have been great company and generous with their offers of help, particularly Han Wang, Erluo Li, Ji-Yong Shin, Ki-Suh Lee, Qi

Huang, Kolbeinn Karlsson, Shannon Joyner, Fan Zhang, and Yunhao Zhang.

I want to thank all the collaborators that I have worked with outside the department. I am grateful to have received the IBM Ph.D. fellowship, and to have spent two wonderful summers at IBM's Thomas J. Watson Research Center, working with my mentors: Christopher C. Young, Sai Zeng, Karin Murthy, and Zhe Zhang. I want to give my special thanks to Zhe Zhang and Nicholas C. Fuller at IBM, Xiaosong Ma and Ben Clay at North Carolina State University, and Xiaozhou Yang at Chinese Academy of Sciences, each of whom were tremendously helpful during the challenging transitions I faced on my Ph.D. journey. I also want to thank Xiaohui Gu and John Wilkes, who gave me guidance during the initial stage of my Ph.D. study.

Last but not least, I want to thank my family. Thanks for my wife Cunfang Shen, who always stood by me no matter how hard it was, and selflessly assisted me through the long journey. I thank my parents, Wenzhang Shen and Ruizhu Shen, for their endless support and prayer. I also thank my son, Arthur, who often dragged me out of my work for a rest and made me laugh. I am sincerely grateful to my family. Without them, I would not have made it this far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources, (e.g., networks, servers, storage, applications, and services,) that can be rapidly provisioned and released with minimal management effort or service provider interaction [93]. The data center hardware and software that supports cloud computing is what we call a *cloud*. We are experiencing an historic shift as more and more computer memory, processing power and applications are moved into the cloud, and organizations become less and less dependent on locally manufactured Information Technology (IT) infrastructures. This allows the long-held dream of *computing as a utility* to come true, which makes software even more attractive as a service and shapes the way IT hardware is designed and purchased [30]. The major benefits of cloud computing include fast resource provisioning, low maintenance cost, and scalability. It is a new paradigm of organization and distribution of computation, storage, and network resources.

Clouds serve multiple users, and mutually untrusted users share cloud resources relying on the cloud platform for *security* isolation — the protection of user computation and resources against unauthorized access, data leakage, and malicious attacks or damages. Additionally, cloud platforms should also provide *flexibility* and *efficiency*. Flexibility refers to the support of user customization on cloud services and resource management policies. Efficiency refers to the capability of the cloud to improve performance and reduce cost of resources such as hardware, energy, man power, money, and time. Achieving security, flexibility, and efficiency is one of the fundamental requirements in cloud in-

frastructures.

Guaranteeing security isolation without sacrificing flexibility and efficiency is challenging. In many systems, security isolation is typically achieved by limiting user customization, which hurts flexibility. Crossing security isolation boundaries generally incurs performance overhead, affecting efficiency. Further, more flexibility implies exposing more control to users, which can lead to weaker security isolation. As an example, operating systems (OSs) such as Linux [43] expose high-level abstractions, such as virtual memory and file systems, and restrict direct access to hardware resources. Virtual memory improves security isolation of different processes, but imposes limitations on applications when they need optimized memory management. Accessing the OS kernel across the kernel isolation boundary is much more expensive than accessing user-level libraries. To support customization and reduce access overhead, Linux allows loading a kernel module and executing user-defined functions within the kernel. However, this breaks most security isolation boundaries enforced by the kernel, and can cause many vulnerabilities. Similarly, containers [110], which refer to user-space instances virtualized by the OS kernel that have separate views on the operating system, support efficient provisioning and execution, but cannot provide sufficient security and flexibility due to the share of underlying OS kernel. The Infrastructure as a Service (IaaS) [93] cloud, a cloud service model that supports users to deploy and run arbitrary software and provides limited control of networking and storage components, exposes high-level abstractions, and poses limitations on how cloud resources can be utilized. As a result, cloud users cannot perform as many operations as they can in a private data center.

This dissertation addresses the following research question: *how to offer flexibility and efficiency as well as strong security in cloud infrastructures?* In particular, we address two important platforms in cloud infrastructures: the containers platform and the Infrastructure as a Service (IaaS) platform. We investigate applying the exokernel approach [57] — a principle of separating protection and management in the design of computation systems. By separating protection and management, containers can be optimized to provide strong security isolation and kernel customization while still supporting efficient execution, and IaaS platforms can enable more flexible and efficient resource management operations without breaking security isolation between users. We first present the scope and background of the dissertation, and then discuss detailed challenges in each platform we address. Based on the challenges, we present our approach and summarize our contributions.

1.1 Scope

1.1.1 Cloud Infrastructure

Cloud computing [30, 93] enables rapid provisioning of shared resources such as computation, storage, and network. It dramatically lowers the cost of IT, and improves resource utilization, availability, reliability, and efficiency. For example, using Amazon Elastic Compute Cloud (EC2) [2], Google Compute Engine [9], or Microsoft Azure [14], it takes seconds or less to allocate computation, storage, and networking to an application. Organizations can thereby focus on core businesses without worrying about infrastructure maintenance. Cloud providers present a "pay-as-you-go" model, so that cloud users are only charged according to actual resource allocation. Users can thereby save costs by

adjusting resources rapidly to meet fluctuating demand. (This is also called "elastic scaling.") Due to these advantages, cloud computing has become a high-demand utility.

A *cloud infrastructure* is the collection of hardware and software that enables cloud computing [93]. It contains both a physical layer and an abstraction layer. The physical layer consists of hardware resources that are necessary to support provided cloud services, and typically includes server, storage and network components. The abstraction layer consists of the software deployed across the physical layer, which manifests essential cloud service models. There are three standard cloud service models as defined by National Institute of Standards and Technology (NIST) [93]:

- *Software as a Service (SaaS)* is a cloud service model that allows users to access applications installed by the provider on the cloud infrastructure.
- *Platform as a Service (PaaS)* is a cloud service model that allows users to deploy onto the cloud infrastructure consumer-created or acquired applications, using programming languages, libraries, services, and tools supported by the provider.
- *Infrastructure as a Service (IaaS)* is a cloud service model that allows users to deploy and run arbitrary software, including operating systems and applications, and provides limited control of networking and storage components.

These models expose different abstractions: SaaS provides a direct software Application Programming Interface (API) or accessible user interface (e.g., web-based email services); PaaS supports programming language and execution

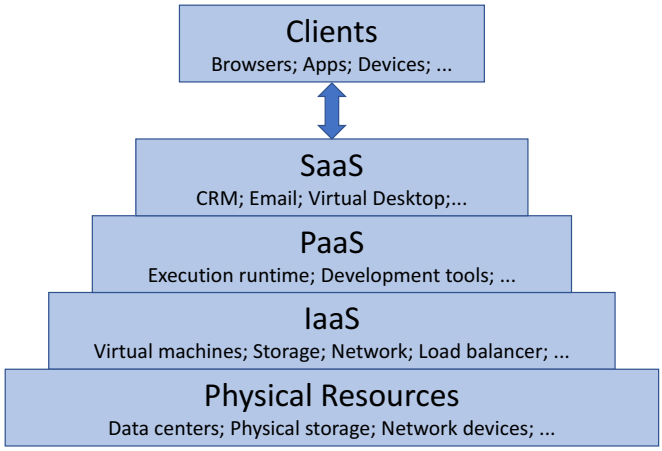


Figure 1.1: A typical cloud infrastructure with layered service models.

runtime; IaaS exposes virtualized computation, storage, and network. In a typical cloud infrastructure, these models often correspond to different layers in the cloud stack, as illustrated in Figure 1.1.

1.1.2 Infrastructure as a Service (IaaS) Platforms

Infrastructure as a Service (IaaS) is a basic cloud service model that allows users to deploy and run arbitrary software including operating systems and applications, and provides limited control of networking and storage components. In the cloud infrastructure, IaaS platforms play a central role in managing and multiplexing shared cloud resources, as illustrated in Figure 1.1. There are many similarities between a traditional computer operating system (OS) and an IaaS platform. An OS manages hardware resources including CPU, memory, disk, and network, and provides abstractions such as processes, threads, and file system. These abstractions greatly facilitate software development, and also provide secure and isolated execution environment for user applications.

Similarly, the major task of an IaaS platform is to efficiently manage data cen-

ter resources, and provide abstractions for efficient and secure resource sharing. The following are key abstractions provided by an IaaS platform:

- **Virtual machine (VM)**: Computation resources are organized in virtual machines (VMs), running on pools of hypervisors such as Xen [33], KVM [82], Hyper-V [117], or VMware ESX/ESXi [53]. Virtualization enables fast provisioning according to users' varying demands, and also ensures security and resource isolation between mutually untrusted users.
- **Software-defined Networking (SDN)**: Network resources are managed with Software-defined Networking (SDN), a technology that allows network administrators to initialize, control, change, and manage network behavior dynamically via open interfaces. Leveraging SDN, IaaS platforms can connect VMs running on different clusters into a Virtual Private Network (VPN), and supports customized routing and firewall rules.
- **Cloud storage**: Storage resources are exposed as cloud storage, in which data is stored in logical pools that span multiple servers, data centers, or locations. Cloud storage can scale instantly, and handle fault-tolerance automatically. It can be used to store unstructured data such as ordinary files, or structured data such as databases and key-value pairs. For structured data, cloud storage also provides interfaces for performing data manipulation and computation.

Major IaaS providers such as Amazon EC2 [2], Google Compute Engine [9], and Microsoft Azure [14] have data centers in different locations world-wide. These data centers are organized in *Regions*. Each region is a separate geographic area, and can have multiple isolated locations known as *Availability*

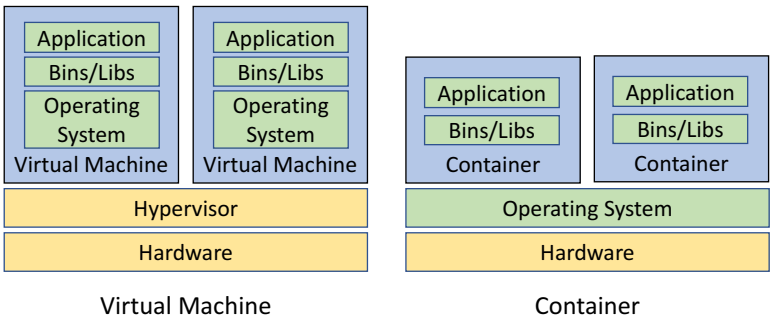


Figure 1.2: Virtual machine vs. container.

Zones. Availability zones in the same region are physically isolated and have independent failures, although they can communicate through low-latency network links. When requesting resources from IaaS platforms, cloud users need to specify the VM configuration (including CPU, memory, network, and storage) and the location (region and availability zone) to place the VM. The resource prices are different in different regions, and communications across availability zones and regions are typically charged.

1.1.3 Containers Platforms

"Containers" is a term that has been used in different contexts, such as resource containers [32], and security containers [101, 125]. With the OS-level virtualization technology [110], a single OS kernel can support multiple isolated user-space instances. Each user-space instance gets a separate view on the file system, network stack, user IDs and groups, and processes. In this dissertation, we use *containers* to refer to such user-space instances virtualized by the OS kernel. As illustrated in Figure 1.2, the key difference between containers and VMs is that containers share the underlying operating system (OS). This eliminates the overhead of running multiple instances of the same operating system, and

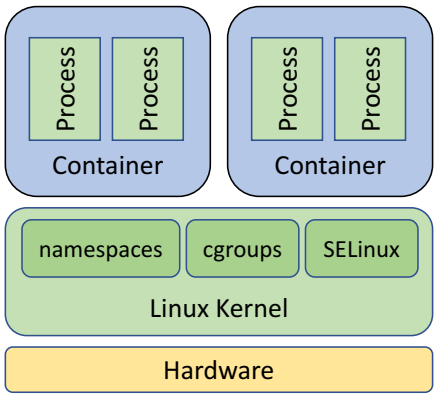


Figure 1.3: Linux container architecture.

makes provisioning and bootstrapping a new container much faster than a traditional VM. Containers have been widely supported in different OSs, such as Linux containers [12, 18], FreeBSD jails [80], and Solaris Zones [101]. In this dissertation we mainly focus on *Linux Containers*, since it is the most mature and widely used container platform.

Linux Containers

Figure 1.3 shows the Linux container architecture. Linux containers rely on a set of technology provided by the Linux kernel [43] to isolate and manage containers:

- **Namespaces**: The Linux kernel provides advanced process isolation by creating separate *namespaces* for containers. A namespace identifies instances of global system resources that are isolated from one another, including mount points, process IDs, and network configurations. Consequently, processes in different namespaces can use the same resource simultaneously without creating a conflict.
- **Control Groups (cgroups)**: The Linux kernel uses *Control Groups (cgroups)*

to group processes for system resource management. A control group is a collection of processes that are bound by the same criteria and associated with a set of parameters or limits. Cgroups can be used to allocate CPU time, system memory, and I/O bandwidth. They can also be used for accounting or prioritizing resource usage of different processes.

- **Security-Enhanced Linux (SELinux)**: *SELinux* is a Linux kernel security module that provides mechanisms for supporting access control security policies. It confines accesses to system resources such as files, devices, system calls that a process can issue, and commands that a user can execute.

Leveraging these in-kernel isolation mechanisms, Linux is able to virtualize multiple OS instances for different groups of processes. Linux also provides management interfaces to interact with the aforementioned kernel components for construction and management of containers. Docker [7] is a software technology for managing and provisioning Linux containers. Docker containers are Linux containers running with *Docker images* — packages of applications with all dependencies including libraries, configuration files, and third party tools. This enables great portability, as an application only needs to be packaged once into a Docker image, and will be able to run anywhere such as public or private clouds. Docker also provides tools to support efficient developing, deploying, and re-using of Docker images.

The Role of Containers in Cloud Infrastructures

Due to their portability and performance, containers have become a key technological component of software distribution and maintenance. As a result, they also play an important role in cloud infrastructures. Major public clouds all

provide native container support. Containers are also one of the key building blocks of a new cloud service model called *serverless computing*. Serverless computing is a cloud computing execution model in which clouds provide runtime environments to execute user programs on-demand, and dynamically manage the allocation of physical resources [92]. In the serverless computing model, user-defined functions are invoked in response to certain events. These user functions are written without the notion of "servers." (Hence the name serverless.) They are charged based on the actual amount of resources consumed by the execution, rather than on pre-purchased units of capacity. Since a function can be invoked millions of times, it is important to provision the runtime efficiently with low overhead. Containers satisfy this requirement well, so most existing serverless computing platforms use containers as the underlying mechanism for packaging and deploying user functions.

1.1.4 The Need for Security, Flexibility, and Efficiency

Cloud computing has become popular due to its flexibility and efficiency. *Flexibility* refers to the support of user customization on cloud services and resource management policies. *Efficiency* refers to the capability of improving performance and reducing cost of resources including hardware, energy, man power, money, and time. However, the model of sharing resources in a centralized place raises security concerns. *Security* means the protection of user computation and resources against unauthorized access, data leakage, and malicious attacks or damages. Cloud providers get access to user data directly, and information can be leaked to other users sharing the same cloud infrastructure. Furthermore, clouds tend to attract more attacks since they maintain valuable data from many companies. For example, Dropbox experienced a data leak-

age in 2014, when over 7 million user passwords were stolen by hackers [10]. Vulnerabilities in cloud architectures such as hardware, hypervisors, and insecure cloud APIs have also been discovered, all of which can potentially cause information leakage [11].

To improve security, cloud providers typically pose limitations on how cloud resources can be consumed and how underlying platforms are implemented, sacrificing flexibility or efficiency. For example, due to the concern of containers isolation, public clouds provide container services by running containers in VMs, at the cost of performance and resource efficiency. Although IaaS platforms provide resources in VMs, users are forced to treat them as physical machines, and advanced resource management techniques enabled by virtualization like VM migration [51] and consolidation [124] are generally not allowed. Many companies choose to run their own private clouds, maintaining physical control over the equipment instead of using off-site machines under third-party control. Although private clouds provide better security and customization support, they lose the efficiency of public clouds and can cost more. The challenge of achieving sufficient security, flexibility, and efficiency is one of the fundamental problems in cloud infrastructures.

1.2 Challenges

The need for security, flexibility, and efficiency presents a challenge to cloud infrastructures. Security isolation is typically achieved by limiting user customization, compromising flexibility. Crossing security isolation boundaries generally incurs performance overhead, affecting efficiency. Increased flexibility can lead to weaker security isolation. In this dissertation we address the

following research question: *How to offer flexibility and efficiency as well as strong security in cloud infrastructures?* In particular, we address two important platforms in cloud infrastructures: the containers and Infrastructure as a Service (IaaS) platforms. In this section we describe challenges inherent to each platform.

1.2.1 Lack of Security and Flexibility in Containers Platforms

Although containers platforms have outstanding efficiency, they do not provide sufficient security and flexibility. Specifically, they have the following problems:

- **Shared kernel attack surface**: All containers on the same host share the same monolithic kernel; if one container is compromised, all containers on the same kernel are put under risk.
- **Kernel incompatibility**: Containers often run into kernel compatibility issues where the application requires some kernel function that is not available on the host kernel.
- **Lack of kernel customization**: Due to concerns about application isolation, containers are generally not allowed to install their own kernel modules, a limitation for applications that requires customized kernels.

An underlying challenge here is that containers are forced to share a kernel over which they do not have any control, which fundamentally limits security and flexibility. An often-used solution is to run containers in VMs, which is how public clouds such as Amazon AWS and Microsoft Azure provide container services. However, VMs sacrifice scalability, performance, and resource efficiency compared to lightweight containers.

1.2.2 Lack of Flexibility and Efficiency in IaaS Platforms

For various reasons including security isolation, public IaaS clouds do not expose many useful APIs enabled by the virtualization infrastructure for efficient resource management, limiting flexibility and efficiency. Given additional management interfaces, applications could easily migrate VMs across availability zones in response to diurnal workloads or changing prices, adjust the resources given to particular VMs in response to load changes, deploy applications across multiple cloud providers for increased fault tolerance, and so on. Control over storage and network routing would further improve customizability for optimal performance. However, there are three major challenges:

- **Heterogeneity in cloud infrastructures**: Different cloud providers use infrastructures based on heterogeneous hypervisors, network, and storage. For example, Amazon EC2, Google Compute Engine, and Microsoft Azure use Xen, KVM, and Hyper-V respectively. They provide different types of VM configurations, and require diverse device drivers and VM images. Furthermore, the network and storage infrastructures of different clouds are typically isolated. There is no widely accepted standard on the cloud API for interoperability. So once an application is deployed into a cloud, it is very hard to move it to another. This problem is generally called *the vendor lock-in problem*.
- **Lack of privileged control**: Although IaaS cloud platforms are typically built with virtualization technology, they only expose high-level abstractions and limit many useful administrative APIs from user access. As a result, cloud users cannot perform administrative tasks as in their own

data center.

- **Lack of common resource management**: Different cloud providers do not have a standard API to communicate with each other. Thus, if an application is deployed to multiple clouds, users have to implement their own resource management platform. It is non-trivial to consider the heterogeneity in computation, network, and storage infrastructures when monitoring and managing applications across different clouds.

1.3 Approach

1.3.1 Separating Protection and Management

The problem of achieving security, flexibility, and efficiency is not unique to cloud infrastructures. In fact, this is a general problem in systems that support *multi-tenancy*: a capability with which a single instance of a software system serves multiple tenants. Here a *tenant* is a group of users or applications that share a common access with specific privileges to the software system. Multi-tenant systems typically provide security isolation between different tenants, and also customization support for each tenant's specific needs. Therefore, they also need to deal with the problem of balancing security, flexibility, and efficiency.

Our approach is inspired by an architecture used to address a similar problem in a multi-tenant system—the operating system (OS). A traditional OS implements high-level abstractions such as virtual memory and inter-process communications. These abstractions make programming easier, but can also limit performance and flexibility. The exokernel architecture [57] was proposed as a

new OS architecture that minimizes the code base running in kernel mode, and provides support of OS customization. Exokernels are small kernels primarily concerned with the sharing of physical resources. Other OS functionality is implemented through so-called *Library OSs* or through shared services running as processes [69]. Security isolation is improved due to a smaller size of Trusted Computing Base (TCB) and attack surface. And user control is improved as well since most OS functionalities are implemented in the same privilege level as applications, and a LibOS is dedicated to a single application for customization.

A fundamental principle in exokernel architecture is *separating protection and management* [78]. This approach is called *the exokernel approach*. Exokernels provide primitives at the lowest level for protection, so that one application cannot easily compromise or interfere with another. Meanwhile, exokernels impose minimal restrictions on resource management, so applications can have great flexibility on implementing their own abstractions and policies for managing resources.

The principle of separating protection and management can be applied generally to multi-tenant systems for improving security, flexibility, and efficiency. With the exokernel approach, a multi-tenant system can be structured with two layers, as illustrated in Figure 1.4. The lower layer, which we call *the protection layer*, multiplexes resources and focuses only on protection and security isolation, while the upper layer, which we call *the management layer*, is composed of management components that are dedicated to each tenant for resource management. Only the protection layer is shared by different tenants, and it is kept relatively simple and small since it does not provide direct application functionality support. All application-oriented features are provided in components of

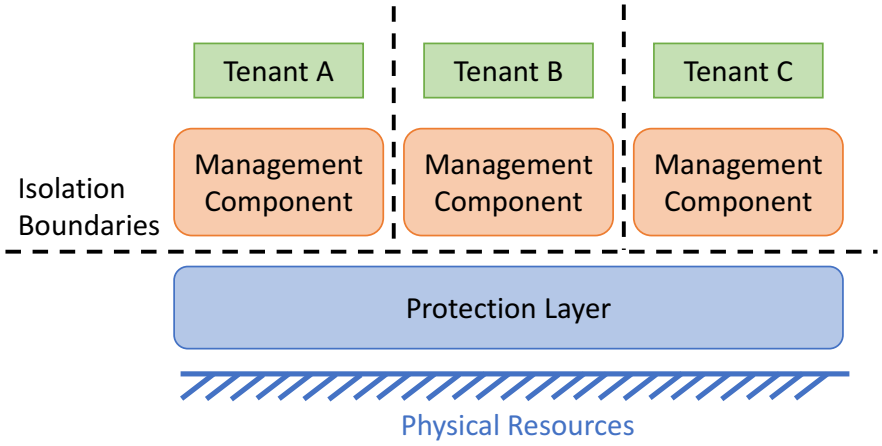


Figure 1.4: Separating protection and management in multi-tenant systems.

the management layer, which are dedicated to each tenant. This architecture brings many benefits:

- **Security**: Security isolation is enforced by a relatively simple and small protection layer, which reduces the complexity of the code base and communication interface. Thus, security isolation is easier to verify and maintain.
- **Flexibility**: Since application-oriented functions are provided in dedicated components of the management layer, applications have the freedom to customize them for their own purpose without affecting others or compromising security isolation.
- **Efficiency**: The dedication of the management layer eliminates the requirement of implementing security isolation, enabling many performance optimization opportunities that are hard to implement when isolation is required. For example, in the exokernel architecture, LibOS can achieve better system call performance than traditional OS by allowing direct function calls to system call handlers.

1.3.2 Limitations

The benefits of the exokernel approach do not come without costs. Since the protection layer does not provide high-level abstractions, its protection policies can only be generic and strict, and it cannot make trade-off without knowing high-level requirements that matter to tenants. Tenants that require more sophisticated security policies related to higher-level abstractions need to implement their own protection mechanisms within management components. This problem can be further complicated by the security policies that involve multiple isolated tenants. In this case, multiple management components need collaboration to maintain a consistent and secure abstraction, which increases system complexity.

Another problem caused by the exokernel approach is that security isolation boundaries among different management components can incur performance overhead. Tenants that collaborate tightly with each other might want to share the same management component, and implement their own security isolation mechanisms. However, it is non-trivial to determine in what situations two tenants should be isolated with different management components, or share the same management component and implement their own isolation mechanisms. Further research is required to provide a general guideline for addressing the trade-off.

1.4 Contributions

The main contribution of this dissertation is applying the exokernel approach to separate protection and management in cloud infrastructures to improve se-

curity, flexibility, and efficiency. By studying two important platforms in cloud infrastructures, the containers and the IaaS platforms, we demonstrate that the exokernel approach can be applied with backward compatibility and incremental deployment. Specifically, we make the following contributions:

First, we show how separating protection and management in containers platforms improves security and flexibility without sacrificing efficiency. We present X-Containers, a new exokernel+LibOS architecture that is fully compatible with Linux containers and provides competitive or superior performance to native Docker Containers as well as other LibOS designs. We untangle the various functions of processes, and in particular use processes for concurrency while proposing X-Containers for security isolation. We compare two different implementations of the X-Container architecture. And finally, we evaluate the efficacy of both implementations of X-Containers and compare them to native Docker and other LibOS implementations (Unikernel [89] and Graphene [115]).

Second, we show how separating protection and management in IaaS platforms improves flexibility and efficiency without sacrificing security. We propose a Library Cloud abstraction for user-level resource management. We show how nested virtualization can be leveraged to handle heterogeneity and implement a Library Cloud without the support of underlying cloud providers. We also propose storage and networking solutions for supporting efficient VM migration in a Library Cloud, and show how they can be implemented efficiently. We evaluate a prototype of the Library Cloud abstraction—the Supercloud, and demonstrate how applications can benefit from the Library Cloud using case studies of geographically shifting workloads and virtual machine consolidation.

1.5 Organization

The rest of this dissertation is organized as follows: In Chapter 2 we present X-Containers, a new containers platform designed with the exokernel approach. In Chapter 3 we present Library Cloud, a new cloud abstraction enabling user-level resource management. We discuss related work in Chapter 4, and future research directions in Chapter 5. Finally, we conclude in Chapter 6.

CHAPTER 2

TOWARDS SECURE, FLEXIBLE, AND EFFICIENT CONTAINERS PLATFORM: X-CONTAINERS

2.1 Introduction

Containers platforms have problems in security and flexibility. If one container is compromised, all containers on the same kernel are put under risk. Due to the concern of application isolation, containers are generally not allowed to install their own kernel modules, a limitation for applications that require customized kernels. Nor can kernels be easily tuned to optimize it for the application at hand.

An oft-used solution is to run each container, packaged with an operating system, in its own virtual machine (VMs). This is how public clouds such as Amazon AWS and Microsoft Azure provide container and serverless services. However, VMs sacrifice scalability, performance, and resource efficiency compared to lightweight containers.

In this chapter we apply the exokernel approach to separate protection and management in containers platforms, and propose a new architecture called *X-Containers*. We demonstrate that the Linux kernel can be modified to serve as a highly efficient LibOS and provide full compatibility to existing applications and kernel modules, while hypervisors can be used as the exokernels to run and isolate them.

Each X-Container hosts an application with a dedicated and possibly customized LibOS based on a Linux kernel. An X-Container can support one or

more user processes, and these run together with the LibOS at the same privilege level. Different processes still have their own address spaces for resource management and compatibility, but they no longer provide secure isolation from one another: processes are used for concurrency, while X-Containers provide isolation.

The X-Container platform automatically optimizes the binary of the application during runtime to improve performance by rewriting costly system calls into much cheaper function calls into the LibOS. X-Containers have 3× raw system call throughput compared to native Docker containers and is competitive or outperforms native containers for other benchmarks.

The X-Container platform also outperforms architectures specialized for containers and serverless services. We have run the `wrk` web benchmark with NGINX on X-Container, Unikernel [89], and Graphene [115]. Under this benchmark, X-Container has comparable performance to Unikernel and about twice the throughput of Graphene. However, when running PHP and MySQL, X-Container has approximately 3× the performance of Unikernel.

While X-Containers borrows much of the software base of Linux, the design and implementation has to address various challenges. For this chapter, we evaluate two contrasting designs. In the first, we run the Linux kernel in user mode alongside user processes on top of Xen. This requires extensive modifications to the Xen hypervisor but does not require special hardware support. Indeed, this design can run both on bare-metal and inside virtual machines in the public cloud. In the second design, we run user processes in kernel mode alongside the Linux kernel exploiting hardware virtualization support—this design can run on any hypervisor and still securely isolates different containers.

In both cases, only the architecture-dependent part of Linux has to be modified.

This chapter makes the following contributions:

- We present X-Containers, a new exokernel-based container architecture that is compatible with Linux containers and that provides competitive or superior performance and isolation to native Docker Containers as well as other LibOS designs. To the best of our knowledge, this is the first architecture that supports secure isolation of containers without sacrificing compatibility, portability, or performance;
- We untangle the various functions of processes, and in particular use processes for concurrency while proposing X-Containers for security isolation;
- We compare two different implementations of the X-Container architecture;
- We evaluate the efficacy of both implementations of X-Containers and compare them to native Docker and other LibOSs (Unikernel and Graphene).

2.2 X-Containers as a New Security Paradigm

2.2.1 Kernel and Process Isolation

Modern operating systems (OS) that support multiple users and processes support various types of isolation, including:

- *Kernel Isolation* ensures that a process cannot compromise the integrity of the kernel nor read confidential information that is kept in the kernel.
- *Process Isolation* ensures that one process cannot easily access or compromise another.

The cost of securing kernel isolation can be significant. System calls to access kernel code are orders of magnitude slower than function calls into a library. Moreover, often data copies are performed in the I/O stack for the sake of eliminating data dependencies between the kernel and user mode code. Meanwhile, there is a trend to push more and more functionality into the OS kernel, and it has become increasingly harder to defend against attacks on the kernel [62]. Modern monolithic OS kernels such as Linux have become a huge code base with complicated services, device drivers, and system call interfaces. It is impractical to verify the security of such a complicated system, and new vulnerabilities are continually being discovered (currently there are more than 500 security vulnerabilities in the Linux kernel [13]).

Process isolation is similarly problematic [75]. For one, this type of isolation typically depends on kernel isolation due to how it is implemented and enforced. But perhaps more importantly, processes are not intended solely for security isolation. They are primarily used for resource sharing and concurrency support, and to support this modern OSs provide interfaces that transcend isolation, including shared memory, shared file systems, signaling, user groups, and debugging hooks. These mechanisms lay out a large attack surface, which causes many vulnerabilities for applications that rely on processes for security isolation.

For example, Apache webservers spawn child processes with the same user

ID. A compromised process can easily access other processes' memory by leveraging the debugging interface (such as `ptrace`) or the `proc` file system. Without careful configuration, a compromised process might also access the shared file system and steal information from configuration files or even databases. Finally, there exists privilege escalation attacks [102] so that a hacker can acquire root privilege to defeat most of the process isolation mechanisms without compromising the kernel.

Indeed, few existing multi-client applications rely on processes for isolating mutually untrusted clients, in particular, they do not dedicate a process to each client. Many do not even use processes at all—popular production systems such as NGINX webserver, Apache Tomcat, MySQL, and MongoDB use event-driven model or multi-threading instead of multi-processing. Multi-process applications such as Apache webserver use a process pool for concurrency instead of security—each process has multiple threads and can be re-used for serving different clients. These applications implement client isolation in the application logic, leveraging mechanisms such as role-based access control, authentication, and encryption.

There are, however, exceptions. The SSH daemon relies on process isolation to isolate different users. Also, if there are multiple applications using the same MySQL daemon on the same kernel, the combination of process isolation and client isolation built into MySQL provides the applications with security in case some applications are compromised—each application poses as a different client to MySQL.

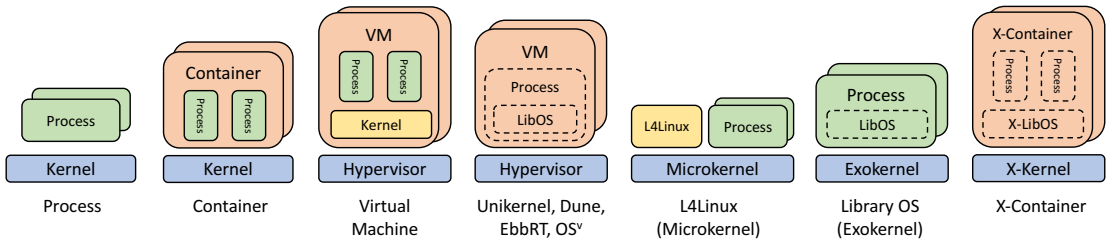


Figure 2.1: Illustration of isolation boundaries in various architectures.

2.2.2 Rethinking the Isolation Boundary

Processes are useful for resource management and concurrency, but ideally security isolation should be decoupled from the process model. Indeed, other isolation mechanisms have been introduced. Figure 2.1 illustrates isolation boundaries in various alternative architectures. Container isolation separates name spaces on top of a kernel such that each container appears to be its own instantiation of that kernel. However, the technology used is essentially the same as process isolation—any isolation that can be accomplished with containers can be accomplished without. It does not solve the problem of the kernel being a large and vulnerable TCB with a large attack surface due to the many available system calls.

From a security point-of-view, running containers in individual VMs, each with a dedicated kernel of its own, is a good solution. The TCB now consists of a relatively small hypervisor with a much smaller attack surface. Unfortunately, overhead is high because of redundant resource consumption and isolation boundaries. Nonetheless, this is now the *de facto* solution for multi-tenant container clouds. In order to deal with the high cost of this solution, more experimental systems such as Unikernel [89], EbbRT [105], OS^v [19], and Dune [34] are lightweight OS kernel alternatives designed to run inside VMs. Unfortunately,

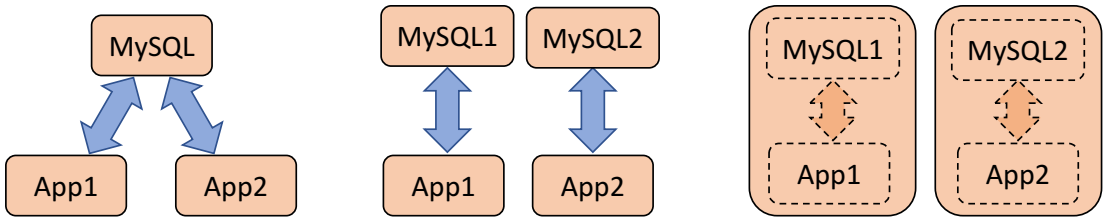
these do not support existing applications well due to lack of binary-level compatibility. Also, typically they can only support single-process applications.

In a microkernel architecture, most of the traditional OS services run in separate user processes alongside application processes. Such architectures can provide binary compatibility. However, because different applications share those OS services, a compromised OS service breaks isolation between the applications—the TCB and attack surface are not reduced. Also, system call overhead tends to be large.

In this chapter we propose the *X-Container* as a paradigm for security isolation between applications. Its architecture is based on the exokernel architecture, having a small kernel attack surface and low system call overhead. An X-Container can have multiple processes—for resource management and concurrency—but not isolation; to isolate users or applications it is necessary to spawn multiple X-Containers. Eliminating isolation within an X-Container can reduce system call overhead to that of a function call.

X-Containers use an "X-LibOS" based on a standard OS Kernel with full binary compatibility. In our implementation, the X-LibOS is derived from Linux, and only required changes to the architecture-dependent part of Linux. The advantages of using a standard kernel are many: Linux is highly optimized and mature, and is being developed by an active community. X-Containers fully leverage this to its fullest, but relies for isolation on a much smaller "X-Kernel." In our implementation, the X-Kernel is derived from Xen.

Different applications should be placed in different X-Containers. To illustrate what exactly we mean by this, consider two applications that each use a



(a) 3 X-Containers (b) 4 X-Containers (c) 2 X-Containers

Figure 2.2: Alternate configurations of two applications that use MySQL.

MySQL database. One option would be to create X-Containers for each application plus a third X-Container dedicated to MySQL (Figure 2.2a). That is, this option treats MySQL as its own isolated application. MySQL internally contains access control logic to securely separate the tables of the two applications.

A more secure configuration would create two instances of MySQL, one for each application, each running in its own X-Container, resulting in four X-Containers total (Figure 2.2b). This would remove reliance on access control logic within the MySQL implementation and thus strictly increase security of the configuration. In addition, this option would provide better customizability, both of the MySQL servers and the operating system kernels that support them.

However, notice how each application has its own MySQL instance. Each application relies on its MySQL instance to store its data durably and respond correctly to queries, while conversely each MySQL instance is dedicated and has nothing lose by being compromised by its own application. Therefore, we can safely deploy only two X-Containers, each containing application logic along with its dedicated MySQL instance (Figure 2.2c). This option provides significantly better performance than the three or four X-Container configurations (see evaluations in Section 2.5.4).

2.2.3 Threat Model of Container Applications

For applications running on X-Containers, or containers in general, we can consider two kinds of threats: external and internal, and these may possibly collude. One type of external threat is posed by messages designed to corrupt the application logic. This threat is countered by application and operating system logic and is identical for standard containers and X-Containers. Another type of external threats may try to break through the isolation barrier of a container. In the case of standard containers, this isolation barrier is provided by the underlying general purpose operating system kernel, which has a large TCB and, due to the large number of system calls, a large attack surface. X-Containers, in contrast, rely for isolation on a small X-Kernel that is dedicated to providing isolation. It has a small TCB and a small number of hypervisor calls that are relatively easy to secure. We believe that X-Containers provide strictly better protection to external threats than standard containers.

Internal threats are created by an application relying on third party libraries or, as illustrated by the MySQL example above, by third party services deployed within the same container. In a Linux container, applications trust Linux to implement isolation between processes owned by different user accounts. X-Containers explicitly do not provide secure isolation between processes in the same container. Applications that rely on secure isolation barriers between processes should either use a standard VM and Linux solution or re-organize the application such that conflicting processes run in different X-Containers. The latter would provide better security but requires more implementation effort.

2.3 Design of X-Containers

2.3.1 Design Goals

Ideally, containers should provide a secure and self-contained environment for running applications. Following are key principles for designing an architecture for running application containers securely:

Self-sufficiency and Customizability: A container should contain all the dependencies of an application. This includes not only libraries, file system layout, and third-party tools, but also the OS kernel. A container should be able to use a customized OS kernel and load its own kernel modules.

Compatibility: A container platform ideally should not require changes to applications. Binary level compatibility allows users to deploy containers immediately without re-writing or even re-compiling their applications.

Isolation with small TCB: Containers should be securely isolated from one another. Although it is necessary to share privileged software to access shared physical resources, that software must be trusted and should be small.

Portability: A key advantage of containers is that they are packaged once and then can be run everywhere, including bare-metal machines and virtualized cloud environments.

Scalability and Efficiency: Application containers should be lightweight and executed with small overhead.

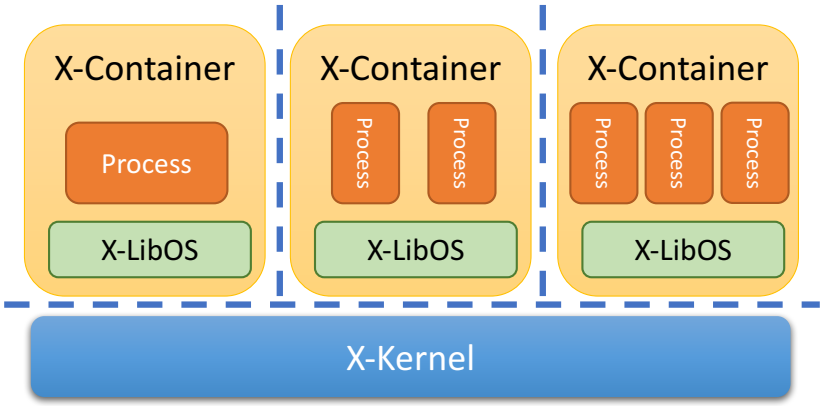


Figure 2.3: The X-Container Architecture. The dashed lines indicate secure isolation barriers.

2.3.2 Architecture

Figure 2.3 illustrates the X-Container architecture. The X-Kernel is an exokernel running in kernel mode. Different applications run in different containers, each with their own customized X-LibOS. An application and its X-LibOS run at the same privilege level for good efficiency.

X-Kernel: The *X-Kernel* is responsible for essential services such as physical resource access and sharing, memory virtualization, and inter-container communication. The Application Binary Interface (ABI) of the X-Kernel is similar to a type-1 VM hypervisor [99]. The X-Kernel comprises a small TCB with a restricted set of system calls, which makes the attack surface much smaller than that of a full-fledged traditional OS kernel.

X-LibOS: The X-LibOS provides the same ABI as a traditional OS kernel to the local application. The X-LibOS provides the application with processes to support convenient resource management and concurrency. Thus they each have their own virtual memory address space by using different page tables, but as

the X-LibOS is not protected from the application, the virtual memory address spaces are not securely isolated.

2.4 Implementation

We use a hypervisor to serve as X-Kernel. We modified a Linux kernel distribution into X-LibOS that allows it to run in the same privilege mode as applications. We explored two possible implementation choices:

1. Para-Virtualized (PV) X-Containers run X-LibOS and applications in user mode. It requires modification of the hypervisor (running in kernel mode), but it does not require special hardware support and can be deployed on bare-metal machines as well as in VMs in public clouds.
2. Hardware Virtualized (HV) X-Containers run X-LibOS and applications in kernel mode. It requires hardware virtualization support, but works with unmodified hypervisors.

For the first implementation choice, we have based the X-Kernel implementation on Xen [33]. It is open source and support of its paravirtualization interface in Linux is mature. For the second implementation choice, we use unmodified Xen with hardware virtualization as the X-Kernel, but other hypervisors could be used as well. For example, we have run HV X-Containers on KVM in Google Compute Engine.

Both implementation choices are of pragmatic interest. The first implementation allows greater control in how X-Containers are managed. For example, it allows running multiple X-Containers securely isolated from one another in the

same VM. Running multiple X-Containers on a single high performance VM will perform better and is more cost effective than running each X-Container in its own, smaller VM. Also, Xen VM management functionalities such as live migration, consolidation, and memory ballooning are supported for PV X-Containers as an added bonus—features not well supported in Linux Containers.

When hardware virtualization is available, the second implementation tends to have better performance. However, in virtualized environments an HV X-Container needs to take over the whole VM unless nested hardware virtualization is supported. VMs in public clouds generally do not expose nested hardware virtualization.

For the experiments, we derived both versions of X-LibOS from Linux kernel 4.4.44. The modifications to the kernel are in the architecture-dependent layer and transparent to other layers in the kernel. We focused on applications running in x86-64 long mode.

Using Linux gives us binary compatibility, but in addition the Linux kernel is also highly customizable. It has hundreds of booting parameters, thousands of compilation configurations, and many fine-grained runtime tuning knobs. Since most kernel functions can be configured as kernel modules and loaded during runtime, a customized Linux kernel can be highly optimized. For example, for applications that run a single thread, such as many event-driven applications, disabling multi-core and Symmetric Multiprocessing (SMP) support can eliminate unnecessary locking and TLB shoot-downs, which greatly improves performance. Depending on the workload, applications can configure the Linux scheduler with different scheduling policies. For many applica-

tions, the potential of the Linux kernel has not been fully exploited due to lack of control over kernel configurations or having to share the kernel with other applications. Turning the Linux kernel into a LibOS and dedicating it to a single application can release all this potential.

2.4.1 Para-Virtualized (PV) X-Containers

Background: Xen Paravirtualization

We implemented PV X-Containers based on the Xen paravirtualization (PV) architecture [33]. The PV architecture enables running multiple concurrent Linux VMs (PV guests or Domain-Us) on the same physical machine without support for hardware-assisted virtualization, but guest kernels require modest changes to work with the underlying hypervisor. In this section we review key technologies in Xen's PV architecture and its limitations on x86-64 platforms.

In the PV architecture, Xen runs in the most privileged mode (the kernel mode) and both guest kernels and user processes run with fewer privileges. All sensitive system instructions that could affect security isolation, such as installing new page tables and changing segment selectors, are executed by Xen. Guest kernels request those services by performing hypercalls, which are validated by Xen before being served. Exceptions and interrupts are virtualized through efficient event channels.

For device I/O, instead of emulating hardware, Xen defines a simpler split driver model. There is a privileged domain (normally Domain-0, the host domain created by Xen during booting) that gets access to hardware devices and multiplexes the device so it can be shared by other Domain-Us. The Domain-

U installs a front-end driver, which is connected to a corresponding back-end driver in Domain-0 through Xen's event channels, and data is transferred using shared memory (asynchronous buffer descriptor rings).

Xen's PV interface has been widely supported by main-line Linux kernels—it was one of the most efficient virtualization technologies on x86-32 platforms. Because there are four different privilege levels for memory segmentation protection (ring-0 to ring-3), we can run Xen, guest kernels, and user processes in different privilege levels for isolation. System calls can be performed without the involvement of Xen. However, the PV architecture faces a fundamental challenge on x86-64 platforms. Due to the elimination of segment protections in x86-64 long mode, we can only run both the guest kernel and user processes in user mode. To protect the guest kernel from user processes, the guest kernel needs to be isolated in another address space. Each system call needs to be forwarded by the Xen hypervisor as a virtual exception, and incurs a switch of page table and TLB flush. This involves significant overhead and is one of the main reasons why 64bit Linux VMs prefer to run fully virtualized nowadays, in hardware virtualization instead of paravirtualization.

Eliminating Kernel Isolation

The PV X-Container architecture is similar to the Xen PV architecture, with one key difference being that the guest kernel (i.e., the X-LibOS) is not isolated from user processes. Instead, they use the same segment selectors and page table privilege level so that kernel access no longer requires a switch between (guest) user mode and (guest) kernel mode, and system calls can be performed with function calls.

This leads to a complication: Xen needs to know whether the CPU is in guest user mode or guest kernel mode for correct syscall forwarding and interrupt delivery. Xen does this using a flag that it can maintain because all user-kernel mode switches are handled by Xen. However, in X-LibOS, with lightweight system calls (Section 2.4.3) and interrupt handling (Section 2.4.1), guest user-kernel mode switches do not involve the X-Kernel anymore. Instead, the X-Kernel determines whether the CPU is executing kernel or process code by checking the location of the current stack pointer. As in the normal Linux memory layout, X-LibOS is mapped into the top half of the virtual memory address space and is shared by all processes. The user process memory is mapped to the lower half of the address space. Thus, the most significant bit in the stack pointer indicates whether it is in guest kernel mode or guest user mode.

In paravirtualized Linux, the "global" bit in the page table is disabled so that switching between different address spaces causes a full TLB flush. This is not needed for X-LibOS, thus the mappings for the X-LibOS and X-Kernel both have the global bit set in the page table. Switching between different processes running on the same X-LibOS do not require a full TLB flush, which greatly improves the performance of address translation. Context switches between different X-Containers do trigger a full TLB flush.

Because the kernel code is no longer protected, kernel routines would not need a dedicated stack if there were only one process. However, the X-LibOS supports multiple processes. Therefore, we still need dedicated kernel stacks in the kernel context, and when performing a system call, a switch from user stack to kernel stack is necessary.

Lightweight Interrupt Handling

In the Xen PV architecture, interrupts are delivered as asynchronous events. There is a variable shared by Xen and the guest kernel that indicates whether there is any event pending. If so, the guest kernel issues a hypercall into Xen to have those events delivered. In the X-Container architecture, the X-LibOS is able to emulate the interrupt stack frame when seeing any pending events and jump directly into interrupt handlers without trapping into the X-Kernel first.

To return from an interrupt handler, an `iret` instruction is used to reset code and stack segments, stack pointer, flags, and instruction pointer together. Interrupts must also be enabled atomically. But in the Xen PV architecture virtual interrupts can only be enabled by writing to a memory location, which cannot be performed atomically with other operations. To guarantee atomicity and security when switching privilege levels, Xen provides a hypercall for implementing `iret`. In the X-Container architecture, we can implement `iret` completely in user mode.

There are two challenges when implementing `iret` in user mode: First, all general registers must be restored before jumping back to the return address, so temporary values such as the stack and instruction pointers can only be saved in memory instead of registers. Second, without issuing hypercalls, virtual interrupts cannot be enabled atomically with other operations. So the code manipulating the temporary values saved in memory must support reentrancy.

There are two cases to consider. When returning to a place running on the kernel mode stack, the X-LibOS pushes temporary registers on the destination stack including the return address, and switches the stack pointer before en-

abling interrupts so preemption is guaranteed to be safe. Then the code jumps to the return address by using a simple `ret` instruction. When returning to the user mode stack, the user mode stack pointer might not be valid, so X-LibOS saves temporary values in the kernel stack for system call handling, enables interrupts, and then executes the `iret` instruction.

Similar to `iret`, the `sysret` instruction, which is used for returning from a system call handler, is optimized without trapping to kernel mode. It is easier to implement `sysret` because it can leverage certain temporary registers.

2.4.2 Hardware Virtualized (HV) X-Containers

The generality of PV X-Containers comes with the cost of performing all sensitive system instructions through hypercalls, including page table manipulations and context switches. Hardware Virtualized (HV) X-Containers eliminate this cost if hardware virtualization support is available.

With hardware virtualization support, X-LibOS can run in kernel mode and execute most privilege instructions directly, which greatly improves performance of page table management and context switches. The major challenge comes from running user processes in kernel mode as well. In addition to modifying the memory and CPU management components in the Linux kernel so that user processes can run in kernel mode, we also need to change how interrupts and exceptions are handled. Because the CPU delivers interrupts and exceptions directly in HV X-Containers, X-Kernel does not have control over how they are handled. The default behavior on the x86 platforms is that no stack switch happens when there is an interrupt or exception in kernel mode. This implies that the interrupt handler can execute on the user stack directly,

which breaks a basic assumption in user code and kernel code: the data on the user stack can be compromised, and much code in the Linux kernel would need to change in order to handle such a situation correctly.

Fortunately, x86-64 introduces a new interrupt stack-switch mechanism, called the Interrupt Stack Table (IST), to force a stack switch on interrupts and exceptions. By specifying a tag in the Interrupt Descriptor Table (IDT), the CPU will switch to a new stack pointer even if the privilege level is not changed. However, nested interrupts become a problem in this case if the same stack pointer is re-used. We solved this problem by specifying a temporary stack pointer in the IST. Right after entering the interrupt handler, we copy the stack frame to the normal kernel stack so that the same stack pointer can be used for nested interrupts.

2.4.3 Lightweight System Calls

In the x86-64 architecture, user mode programs perform system calls using the `syscall` instruction, which transfers control to a routine in kernel mode. The X-Kernel immediately transfers control to the X-LibOS, guaranteeing binary level compatibility so that existing applications can run on the X-LibOS without any modification.

Because the X-LibOS and the process both run in the same privilege level, it is more efficient to invoke system call handlers directly. However, a challenge arises from the setting of the `GS` segment. The Linux kernel stores per-CPU variables in the `GS` segment. This segment is set by a `swapgs` instruction on entering the kernel for every system call, and re-set before returning to the user program. Unfortunately, the `swapgs` instruction is only valid in kernel mode.

It is possible to change the per-CPU variable placement by avoiding the use of segmentation. But to keep the change to Linux kernel minimal, we instead disable the `GS` segment switch when entering or leaving the X-LibOS, keeping the `GS` segment valid all the time. x86-64 applications might use the `FS` segment for thread local storage, but the `GS` segment is typically not touched. We have not yet seen any application that needs a customized `GS` segment.

Another challenge comes from the mechanism of enabling lightweight system calls. X-LibOS stores a *system call entry table* in the `vsyscall` page, which is mapped to a fixed virtual memory address in every process. Updating X-LibOS will not affect the location of the system call entry table. Using this entry table, applications can optimize their libraries and binaries for X-Containers by patching the source code to change system calls into function calls, as most existing LibOSs do. But it significantly increases deployment complexity, and it cannot handle third-party tools and libraries that do not have source code available. To avoid re-writing or re-compiling the application, we implemented an online Automatic Binary Optimization Module (ABOM) in the X-Kernel for PV X-containers and in X-LibOS for HV X-Containers. It replaces `syscall` instructions with function calls automatically on the fly. There are many challenges for in-place binary replacement:

1. **Binary level equivalence**: the total length of the patched instructions cannot be changed, and the program must perform exactly the same functions even when the application code jumps into the middle of a patched block.
2. **Position-independence**: we can only call an absolute address stored in memory or a register, instead of a relative address displacement, because libraries such as `glibc` are loaded in different locations for different pro-

cesses.

3. **Minimum performance impact**: it is impractical to scan the whole binary when loading the application or during runtime.
4. **Handling read-only pages**: most of the binary code is mapped read-only in memory. The binary replacement cannot trigger copy-on-write mechanisms in X-LibOS, otherwise potentially many copies of the same memory page may be created for different processes.
5. **Concurrency safety**: the same piece of code can be shared by multiple CPUs running different threads or processes. The replacement should be done atomically without affecting or stopping other CPUs.
6. **Swapping safety**: memory swapping may happen during the replacement. The system should be able to detect and handle it correctly without compromising memory or causing high performance overhead.

ABOM performs binary replacement on the fly when receiving a syscall request from user processes, avoiding scanning the entire binary file. Before forwarding the syscall request, ABOM checks the binary around the syscall instruction and see if it matches any pattern that it recognizes. If it does, ABOM temporarily disables the write-protection bit in the `CR-0` register, so that code running in kernel mode can change any memory page even if it is mapped read-only in the page table. ABOM then performs the binary patch with atomic `cmpxchg` instructions. Since each `cmpxchg` instruction can handle at most eight bytes, if we need to modify more than eight bytes, we need to make sure that any intermediate state of the binary is still valid for the sake of concurrency safety. The patch is mostly transparent to X-LibOS, except that the page table dirty bit will be set for read-only pages. X-LibOS can choose to either ignore

those dirty pages, or flush them to disk so that the same patch is not needed in the future.

A bigger problem is to handle swapping safety. Especially in PV X-Containers, although the decision of memory swapping is made by X-LibOS, all page table manipulations are done through hypercalls in the X-Kernel. X-Kernel can lock the page table to prevent swapping during binary replacement, but this can cause higher performance overhead. We ended up implementing the binary replacement as follows: binary replacement runs in the context of system calls, so if the target page is swapped out right before the replacement, writing to the page will trigger a page fault. ABOM captures this page fault and continues to forward the system call without propagating it to the page fault handler of X-LibOS. ABOM will try to patch the same location next time when it is executed.

Figure 2.4 illustrates two patterns of binary code that ABOM recognizes. To perform a system call, programs typically set the system call number in the `rax` or `eax` register with a `mov` instruction, and then execute the `syscall` instruction. The `syscall` instruction is two bytes, and the `mov` instruction is 5 or 7 bytes depending on the size of operands. We replace these two instructions with a single `call` instruction with an absolute address stored in memory, which can be implemented with 7 bytes. The memory address of the entry points is retrieved from the system call entry table stored in the `vsyscall` page. The binary replacement only need to be performed once for each place.

With 7-byte replacements, we merge two instructions into one. There is a rare case that the program jumps directly to the location of the original `syscall` instruction after setting the `rax` register somewhere else. After the replacement,

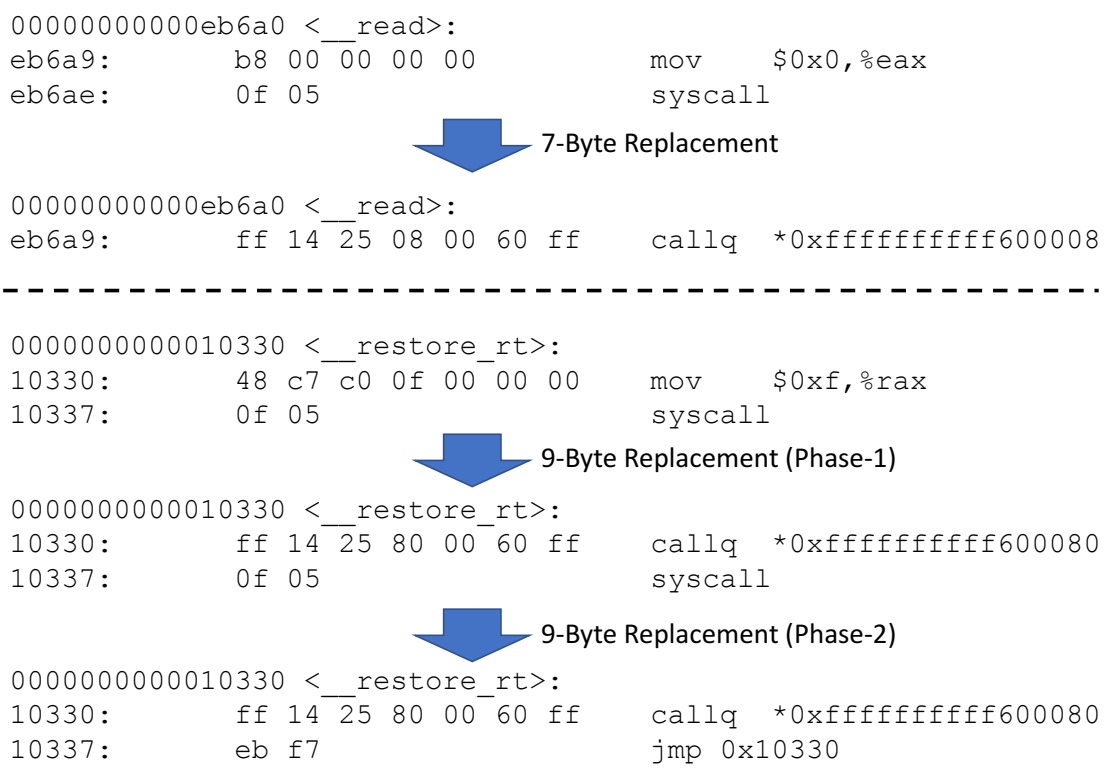


Figure 2.4: Examples of binary replacement.

this will cause a jump into the last two bytes of our `call` instruction, which are always "`0x60 0xff`". These two bytes cause an invalid opcode trap into the X-Kernel (PV) or X-LibOS (HV). To provide binary level equivalence, we add a special trap handler in the X-Kernel (only in the case of PV) and X-LibOS to fix the trap by moving the instruction pointer backward to the beginning of the call instruction. We have only seen this triggered during the boot time of some operating systems.

9-byte replacements are performed in two phases, each one of them generates results equivalent to the original binary. Since the `mov` instruction takes 7 bytes, we replace it directly with a call into the syscall handler. We can leave the original syscall instruction unchanged, just in case the program jumps directly to it. But we further optimize it with a jump into the previous `call` instruc-

tion. The syscall handler in X-LibOS will check if the instruction on the return address is either a `syscall` or a specific `jmp` to the `call` instruction again. If it is, the syscall handler modifies the return address to skip this instruction.

Our online binary replacement solution only handles the case when the `syscall` instruction immediately follows a `mov` instruction. For more complicated cases, it is possible to inject some code into the binary and re-direct a bigger chunk of code. We also provide a tool to do it offline. For most standard libraries such as `glibc`, the default system call wrappers typically use the pattern illustrated in Figure 2.4. So our current solution is sufficient for optimizing most system call wrappers on the critical path.

2.4.4 Lightweight Bootstrapping of Docker Images

X-Containers do not have a VM disk image and do not go through the same bootstrapping phase that a VM does. To bootstrap an X-Container, the X-Kernel loads an X-LibOS with a special bootloader into memory and jumps to the entry point of the X-LibOS directly. The bootloader initializes virtual devices, including setting IP addresses, and then spawns the process in the container without running any unnecessary services. The first process in the container can fork additional processes if necessary. In addition, HV X-LibOS can be also loaded by GRUB [8] with the special bootloader without the help of underlying hypervisors. This approach makes X-Containers smaller than a typical VM and faster to boot. For example, we are able to spawn a new Ubuntu-16 X-Container with a single `bash` process within three seconds, with a memory size of 64MB.

Because X-Containers support binary level compatibility, we can run any existing Docker image without modification. We connect our X-Container archi-

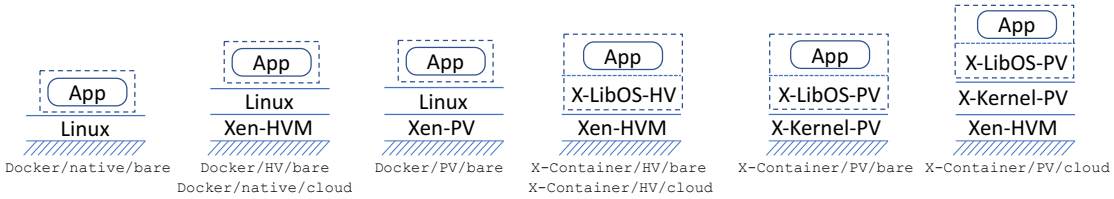


Figure 2.5: Software stacks used in the evaluation. A dashed box indicates a Docker container or X-Container. A solid line indicates an isolation boundary between privilege levels. A dotted line indicates a library interface.

tecture to the Docker platform with a *Docker Wrapper*. An unmodified Docker engine running in the Host X-Container is used to pull and build Docker images. We use `devicemapper` as the storage driver, which stores different layers of Docker images as thin-provisioned copy-on-write snapshot devices. The Docker Wrapper then retrieves the meta-data from Docker, creates the thin block device and connects it to a new X-Container. The processes in the container are then spawned with a dedicated X-LibOS.

2.5 Evaluation

In this section we address the following questions:

- What is the performance overhead of X-Containers, and how does it compare to native Docker both on bare metal and in the cloud?
- How does the performance of X-Containers compare to other LibOS designs?
- How does the scalability of X-Containers compare to native Docker Containers?
- How can kernel customization benefit performance?

2.5.1 Experiment Setup

We conducted experiments on both bare-metal machines and on VMs in public clouds. For bare-metal experiments, we used four Dell PowerEdge R720 servers (two 2.9 GHz Intel Xeon E5-2690 CPUs, 16 cores, 32 threads, 96GB memory, 4TB disk), connected to one 10Gbit switch. For the cloud environment, we ran experiments in four VMs in the Amazon EC2 North Virginia region (`m3.xlarge` instances, 2 CPU cores, 4 threads, 15GB memory, and 2×40GB SSD storage).

As a baseline we ran a Docker container platform both on bare-metal and in an Amazon HV machine. We call these two configurations `Docker/native/bare` and `Docker/native/cloud` respectively. We contrasted their performance against Docker containers running in individual Xen HV and PV Domain-U VMs and against X-Containers. This led to six additional configurations: `Docker/HV/bare`, `Docker/PV/bare`, `X-Container/HV/bare`, `X-Container/PV/bare`, `X-Container/HV/cloud`, and `X-Container/PV/cloud`. Figure 2.5 illustrates the various software stacks. Note that of these eight configurations, three run in the cloud and five on bare-metal.

The host (either a physical machine or an Amazon EC2 instance) running native Docker had Ubuntu 16.04-LTS installed with Docker engine 17.03.0-ce and Linux kernel 4.4.44. The host running Xen VMs had CentOS-6 installed in Domain-0, and Ubuntu 16.04-LTS in Domain-Us with Docker engine 17.03.0-ce, Linux kernel 4.4.44, and Xen 4.2. The host running X-Containers used X-LibOS based on Linux kernel 4.4.44, and CentOS-6 as the Host X-Container. Docker containers used the default NUMA-aware Linux scheduler, with IRQ-balance

service turned on. The Domain-0 and Host X-Container were configured with dedicated CPU cores, and we manually balanced IRQ to different cores. Other VMs or X-Containers were evenly distributed to other CPU cores according to the NUMA placement.

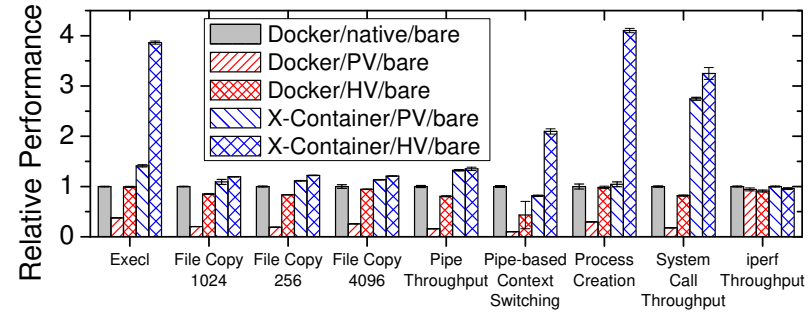
For each set of experiments, the same Docker image was used. The Docker engines were all configured with `device-mapper` storage drivers. When running network benchmarks that involved a client or a server, a separate machine or VM was used.

Because applications running in X-Containers have full control over X-LibOS, they can disable Symmetric Multiprocessing (SMP) and multi-core support when there is only a single thread busy. This optimization can improve performance significantly in many cases, eliminating concurrency management and TLB shoot-down. Applications running in Docker containers cannot do this kind of optimization because it requires root privilege. In the micro- and macro-benchmarks that follow, we performed both single-process and multi-process tests. We disabled SMP support in X-LibOS for single-process cases.

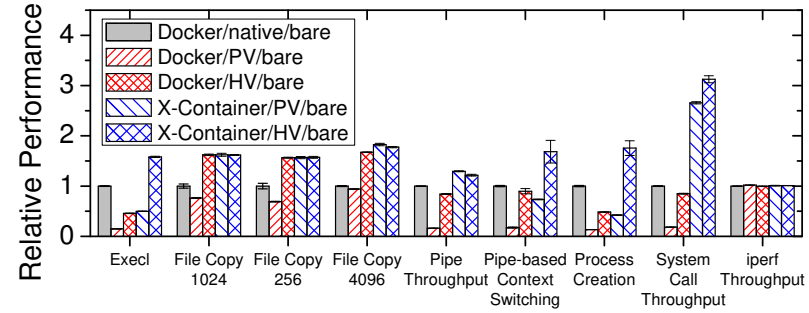
For most experiments we report the average of five runs and also show standard deviation.

2.5.2 Microbenchmarks

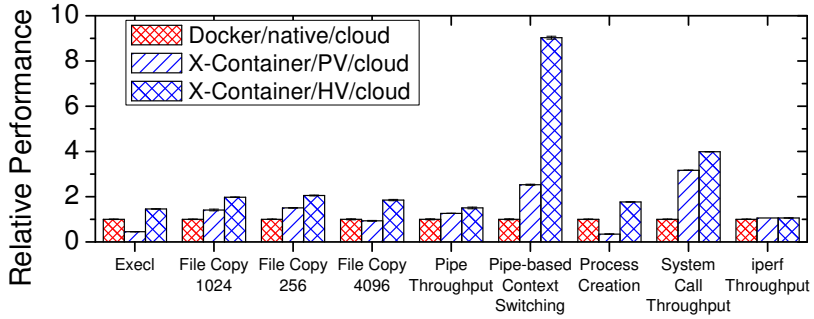
We evaluated the performance of X-Containers with a set of microbenchmarks. We started with an Ubuntu16 Docker image, and ran `UnixBench` and `iperf` in it. The *Execl* benchmark measures the speed of the `exec` system call, which overlays a new binary on the current process. The *File Copy* benchmarks test



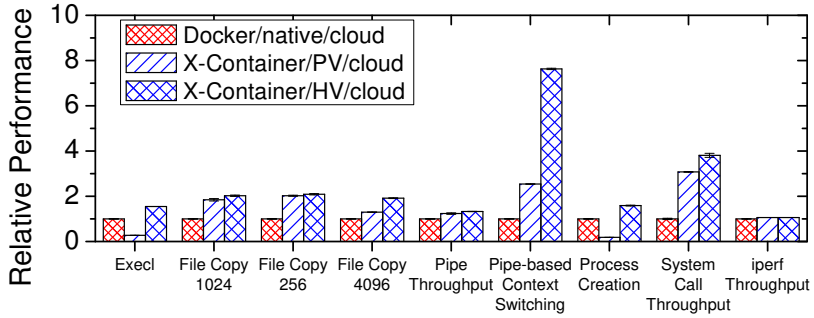
(a) Bare-metal Single



(b) Bare-metal Concurrent



(c) Amazon EC2 Single



(d) Amazon EC2 Concurrent

Figure 2.6: Relative performance of microbenchmarks.

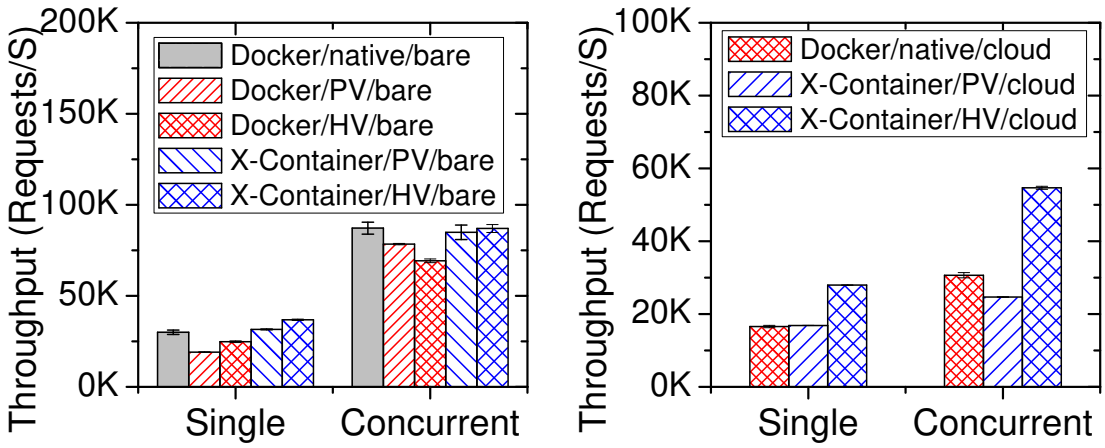
the throughput of copying files with different buffer sizes. The *Pipe Throughput* benchmark measures the throughput of reading and writing in a pipe. The *Pipe-based Context Switching* benchmark tests the speed of two processes communicating with a pipe. The *Process Creation* benchmark measures the performance of spawning new processes with the `fork` system call. The *System Call* benchmark tests the speed of issuing a series of system calls including `dup`, `close`, `getpid`, `getuid`, and `umask`. Finally, the *iperf* benchmark tests the performance of TCP transfer. For concurrent tests, we ran 4 copies in bare-metal experiments, and 2 copies in Amazon EC2 experiments since the EC2 instance had only two CPU cores.

Figure 2.6 shows the relative performance normalized according to `Docker/Native` (higher is better). X-Containers have significantly higher system call throughput because we turned system calls into lightweight function calls. For single-process benchmarks we optimized X-LibOS by disabling SMP support, and as a result X-Containers significantly outperform Docker. `X-Container/PV` had significant overheads compared to `Docker/Native` in process creation and context switching, especially in virtualized environments such as Amazon EC2. This is because process creation and context switches involves many page table operations, which must be done in the X-Kernel. `X-Container/HV` removes this overhead and achieved better performance than both `Docker/native` and `Docker/HV/bare`. (`Docker/HV/bare` achieves better performance than `Docker/native/bare` in file copy benchmarks because there is an extra layer of disk caching.)

2.5.3 Macrobenchmarks

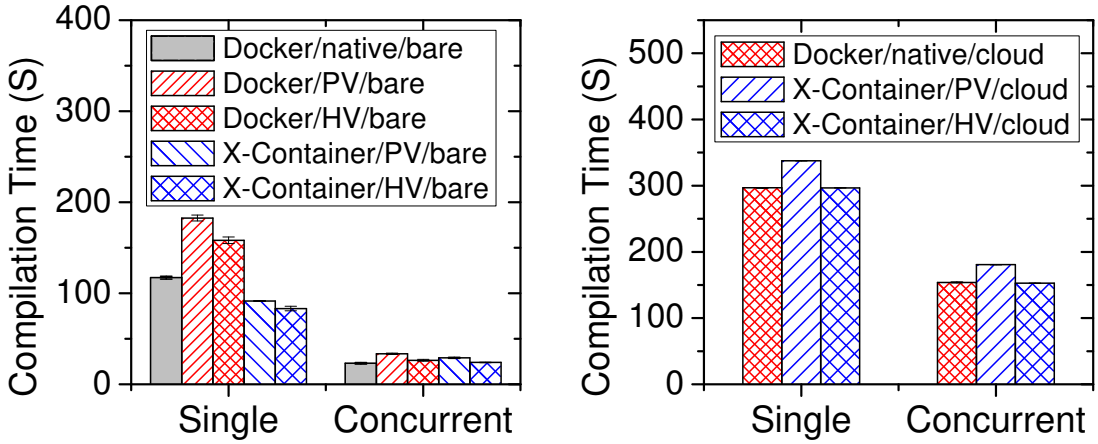
We evaluated the performance of X-Containers with two macrobenchmarks: the NGINX web server and kernel compilation. For the NGINX server, we ran Docker image `NGINX:1.11` on all platforms. We used the `wrk` benchmark to test the throughput of the NGINX server with both single and multiple worker processes. The `wrk` client started 10 threads and 100 connections for each worker process in the NGINX server. On bare-metal machines, Docker containers and X-Containers used a bridged network and can be connected to clients directly. On Amazon EC2, they used a private network with port forwarding. Note that `X-Container/HV/cloud` took over the whole HVM in EC2, so it got access to the network without port forwarding. For kernel compilation tests, we used the Ubuntu-16.04 Docker image and installed compilation tools in it. We compiled the latest 4.10 Linux kernel with the "tiny" configuration. Concurrent tests are performed by running 4 parallel jobs in bare-metal experiments and 2 parallel jobs in Amazon EC2 experiments.

Figure 2.7 shows the NGINX web server throughput measured on bare-metal machines and Amazon EC2. X-Containers consistently outperformed Docker containers inside Xen VMs due to kernel customization and reduced system call overhead. When running a single worker process, `X-Container/PV/bare` and `X-Container/HV/bare` were further optimized by disabling SMP support and achieved 5% and 23% higher throughput than `Docker/native/bare` containers respectively. When running concurrent worker processes on bare-metal, the performance of X-Containers was comparable to `Docker/native/bare` containers. In Amazon EC2, `X-Containers/HV/cloud` achieved 69% to 78% higher through-



(a) Bare-metal (b) Amazon EC2

Figure 2.7: NGINX web server throughput.



(a) Bare-metal (b) Amazon EC2

Figure 2.8: Kernel compilation time. (Lower is better.)

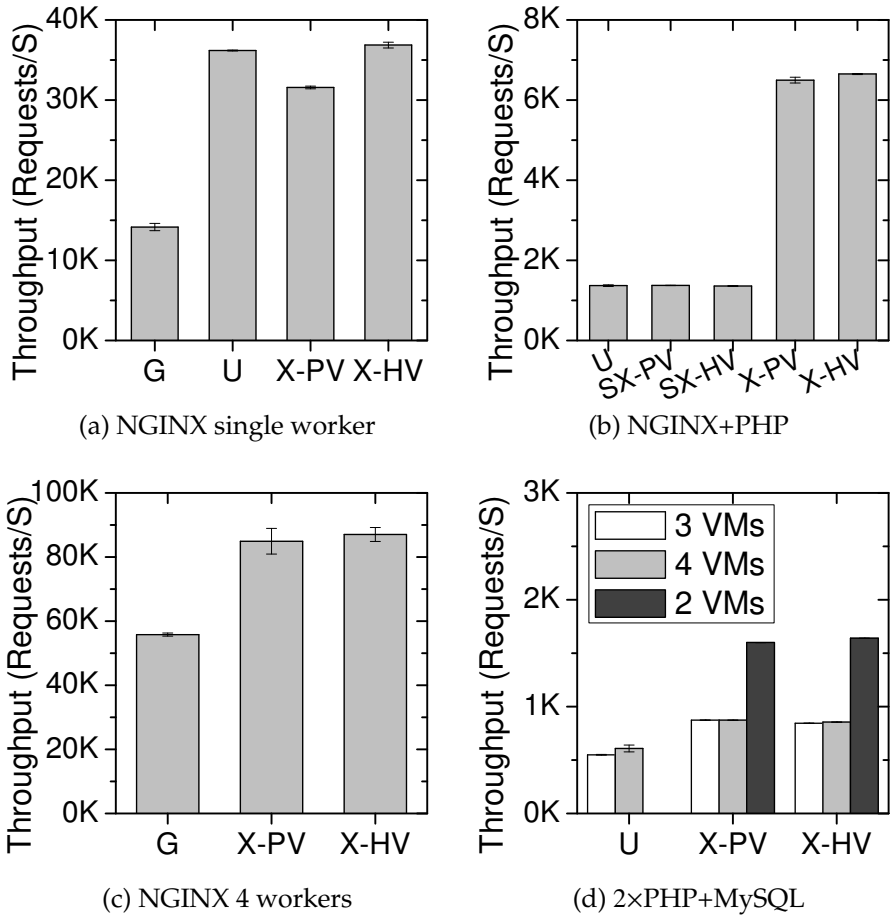
put than `Docker/native/cloud` since it took over the whole HVM and ran without port forwarding. Due to context switch overhead, `X-Containers/PV/cloud` had a 20% performance loss in concurrent tests compared to `Docker/native/cloud`. This result shows that for network I/O intensive workloads, X-Containers perform better than VMs and in many cases even better than native Docker containers.

Figure 2.8 shows the kernel compilation time on bare-metal machines and Amazon EC2 instances. Lower is better. Similar to NGINX experiments, single-process X-Containers on bare-metal machines performed significantly better than Docker containers running natively or in VMs. We did not see a similar improvement in Amazon EC2, which we suspect is due to another layer of I/O scheduling. PV X-Containers performance suffered a bit because of the high overhead of page table management in paravirtualized environments, slowing down operations such as `fork` and `exec`. This result shows that, for CPU intensive workloads, the performance benefit we can get from lighter-weight system calls is limited, but a performance boost is still possible by kernel customization.

2.5.4 Comparing X-Containers to Unikernel and Graphene

We compared X-Containers to Graphene and Unikernel. We ran the `wrk` benchmark with NGINX webserver, PHP, and MySQL database on bare-metal machines. Graphene ran on Linux with Ubuntu-16.04, and was compiled *without* the security isolation module (which should improve its performance). For Unikernel, we used *Rumprun* [22] because it can run those applications with minor patches (running with MirageOS [89] requires rewriting the entire application with OCaml). Unikernel does not support running in Xen HV, so we only tested it with PV mode.

Figure 2.9a compares the throughput of NGINX webserver serving static webpages with a single worker process. As there is only one NGINX server process running, we optimized X-Containers by disabling SMP. X-Containers achieved throughput comparable to Unikernel, and over twice that



(a) NGINX single worker

(b) NGINX+PHP

(c) NGINX 4 workers

(d) 2×PHP+MySQL

Figure 2.9: Performance comparison to Unikernel and Graphene (G: Graphene; U: Unikernel; X-PV: X-Container PV; X-HV: X-Container HV; SX-PV: Separated X-Container PV; SX-HV: Separated X-Container HV).

of Graphene.

Figure 2.9b shows throughput for an NGINX webserver and a PHP CGI server. Graphene does not support PHP CGI server, so we only compared to Unikernel. Because Unikernel cannot run multiple processes, we ran two Unikernel VMs connected with a virtual network, and compared it to a similar setup with two X-Containers. We found that the throughput was very close because the major bottleneck is due to network I/O. In contrast, an X-Container can run a Docker image with an NGINX process and a PHP process connected

with a local socket. This significantly helps performance: X-Container throughput was about five times that of the Unikernel setup.

Figure 2.9c shows a case of running 4 worker processes of a single NGINX webserver. This is not supported by Unikernel, so we only compared to Graphene. In this case, X-Container outperformed Graphene by more than 50%. The performance of Graphene was limited because in Graphene multiple processes use IPC calls to coordinate access to a shared POSIX library, which causes significant overhead.

We evaluated the scenario illustrated in Section 2.2.2, where two PHP CGI servers are connected to MySQL databases. We enabled the built-in webserver of PHP, and used the `wrk` client to access a page that issued requests to the database (with equal probability for read and write). As shown in Figure 2.2, the PHP servers can either share the database or have separate ones. Graphene does not support PHP CGI server, so we only compared to Unikernel. Figure 2.9d shows the total throughput of two PHP servers with different configurations. All VMs were running single process with one CPU core. With 3-VM and 4-VM configurations, X-Containers outperformed Unikernel by more than 40%. We believe this is because the Linux kernel is better optimized than the Rumprun kernel. Further, X-Container supports running PHP and MySQL in a single container, which is not possible for Unikernel. This convenience also significantly helps performance: X-Container throughput was about three times that of the Unikernel setup.

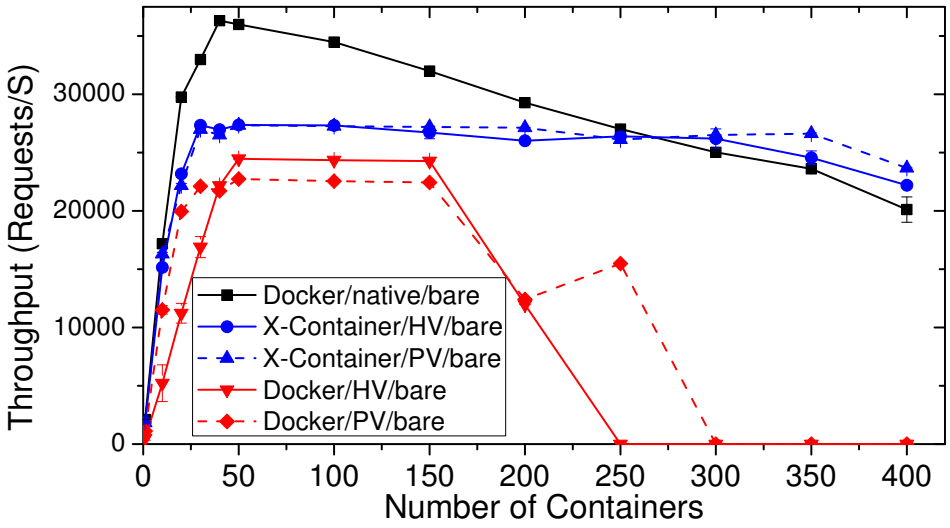


Figure 2.10: Scalability.

2.5.5 Scalability Evaluations

We evaluated the scalability of the X-Container architecture by running up to 400 containers on one physical machine. For this experiment we used an NGINX server with a PHP-FPM engine. We used the `webdevops/PHP-NGINX` Docker image and configured NGINX and PHP-FPM with a single worker process. We ran the `wrk` benchmark to measure the total throughput of all containers. Each container has a dedicated `wrk` thread with 5 concurrent connections—thus the total number of `wrk` threads and concurrent connections increases linearly with the number of containers.

Each X-Container was configured with 1 virtual CPU and 128MB memory, with X-LibOS optimized by disabling SMP support.[1] For `Docker/HV/bare` and `Docker/PV/bare`, each Xen VM was assigned 1 virtual CPU and 512MB memory (512MB is the recommended minimum size for Ubuntu-16 OS). How-

[1]X-Containers also work with 64MB memory, but for this experiment 128MB memory size is sufficiently small to boot 400 X-Containers.

ever, because the physical machine only has 96GB memory, we had to change the memory size of the VMs to 256MB when starting more than 200 VMs. We found that the VMs could still boot but the network stacks started dropping packets. We were not able to boot more than 250 PV instances or more than 200 HV instances correctly on Xen.

Figure 2.10 shows the aggregated throughput of all bare-metal configurations. We can see that `Docker/native/bare` containers achieved higher throughput for small numbers of containers. This is because context switching between Docker containers is cheaper than between X-Containers and between Xen VMs. However, as the number of containers increased, the performance of Docker containers dropped faster. This is because each NGINX+PHP container ran 4 processes: with N containers, the Linux kernel running Docker containers was scheduling $4N$ processes, while X-Kernel was scheduling N virtual CPUs, each running 4 processes. This hierarchical scheduling turned out to be a more scalable way of scheduling many containers together, and with $N = 400$ `X-Container/PV/bare` outperformed `Docker/native/bare` by 18%.

2.5.6 Performance Benefits of Kernel Customization

X-Container enables application containers that require customized kernel modules. For example, X-Containers can use software RDMA (both Soft-iwarp and Soft-ROCE) applications. In Docker environments such modules require root privilege and exposes the host network to the container directly, raising security concerns.

In this section we present a case study of kernel customization in X-Containers that illustrates a performance benefit. We tested a scenario with

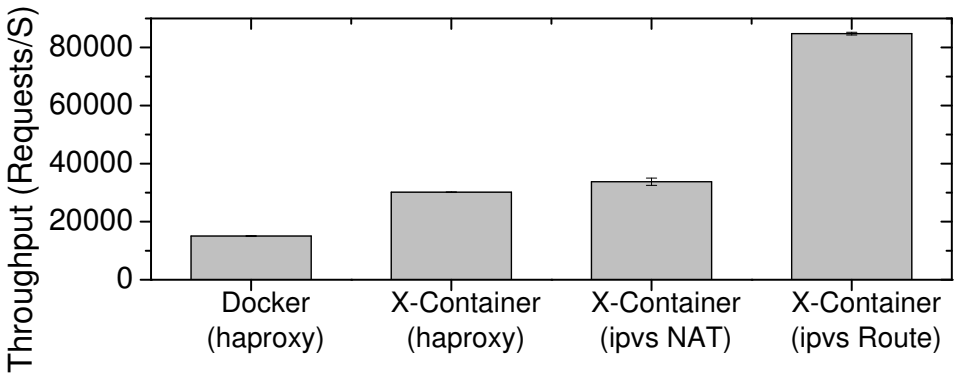


Figure 2.11: Kernel-level load balancing.

three NGINX web servers and a load balancer. The NGINX web servers are each configured to use one worker process. Docker platforms typically use a user-level load balancer such as *HAProxy*. HAProxy is a single threaded, event-driver proxy server widely deployed in production systems. X-Containers supports HAProxy, but can also use kernel-level load balancing solutions such as IPVS (IP Virtual Server). IPVS requires inserting new kernel modules and changing iptable and ARP table rules, which is not possible in Docker containers without root privilege and access to the host network.

In this experiment, we used the `HAProxy:1.7.5` Docker image. The load balancer and NGINX servers were running on the same physical machine. We configured each X-Container with a single virtual CPU, with SMP supported turned off in X-LibOS for optimized performance. We used the `wrk` workload generator and measured total throughput.

Figure 2.11 compares various configurations. The X-Container platform with HAProxy achieved twice the throughput of the Docker container platform. With IPVS kernel level load balancing using NAT mode, X-Containers further improve throughput by 12%. In this case the load balancer was the bottleneck because it served as both web front-end and NAT server. IPVS supports an-

other load balancing mode called "direct routing." With direct routing, the load balancer only needs to forward requests to backend servers while responses from backend servers are routed directly to clients. This requires changing iptable rules and inserting kernel modules both in the load balancer and NGINX servers. With direct routing mode, the bottleneck shifted to the NGINX servers, and total throughput improved by another factor of 1.5.

2.6 Summary

In this chapter we demonstrated how we can use an exokernel architecture to run unmodified container distributions efficiently and securely, both on bare metal and in the cloud. One important insight is that the Linux kernel makes an excellent Library OS when modified to run at the same privilege level as the container. Linux is highly optimized and already supports customization. System calls can be dynamically rewritten into function calls to reduce unnecessary abstraction overheads.

CHAPTER 3

TOWARDS SECURE, FLEXIBLE, AND EFFICIENT IAAS PLATFORM: LIBRARY CLOUD

3.1 Introduction

Existing IaaS cloud platforms lack sufficient support for flexibility and efficiency, violating a basic systems design principle as formulated by Butler Lampson: *Don't Hide Power* [85], or, in other words, do not hide desirable interfaces. Cloud providers hide many powerful management interfaces, including VM placement and migration, CPU capping, memory ballooning, page sharing, I/O throttling, and network routing. Without such interfaces, applications are severely limited in the ways they can optimally configure resources or respond to dynamically shifting workloads.

In this chapter, we apply the exokernel approach to separate protection and management in IaaS platforms, and introduce the *Library Cloud* abstraction. In a Library Cloud, cloud providers are only needed to provide basic resources for computation, storage, and networking. A Library Cloud implements an entire cloud stack on top of these resources, as it were, in user space. Moreover, these resources can be allocated at multiple cloud providers.

Like a private IaaS cloud, a Library Cloud supports traditional cloud services for allocating virtual machines. But a Library Cloud also provides all the administrative APIs that are not available to users of public cloud providers. Because a Library Cloud supports management operations such as user-level migration and memory consolidation, it enables application solutions not easily achieved with disjoint cloud providers or even federated clouds based on

standard interfaces. Two examples of this are management of geographically shifting workloads and transparent consolidation of virtual machine resources. We describe and evaluate these examples in this chapter.

We built a prototype Library Cloud that we call the *Supercloud*. Building the Supercloud, we had to overcome various challenges related to computation, storage, networking, and management. Although some cloud providers such as Amazon allow customers to rent dedicated physical servers, most cloud providers do not expose physical resources directly. Supercloud leverages nested virtualization technology [122], eliminating the need for VM management support from underlying providers. Xen-Blanket unifies machine virtualization on different platforms and provides the same interfaces that are supported in the Xen hypervisor.

We designed and implemented a new distributed storage system optimized for wide-area cross-provider VM migration. It decouples providing strong consistency from update propagation, minimizing overhead and optimizing performance. VMs run in a high performance Software-Defined Network (SDN) built with Open-vSwitch and VXLAN tunnels crossing cloud boundaries. We also designed and evaluated various solutions to deal with migrating services that use public IP addresses. The Supercloud runs the OpenStack platform and appears to users as a single private OpenStack cloud.

This chapter makes the following contributions:

- We propose a Library Cloud abstraction for user-level resource management.
- We show how nested-virtualization can be leveraged to handle hetero-

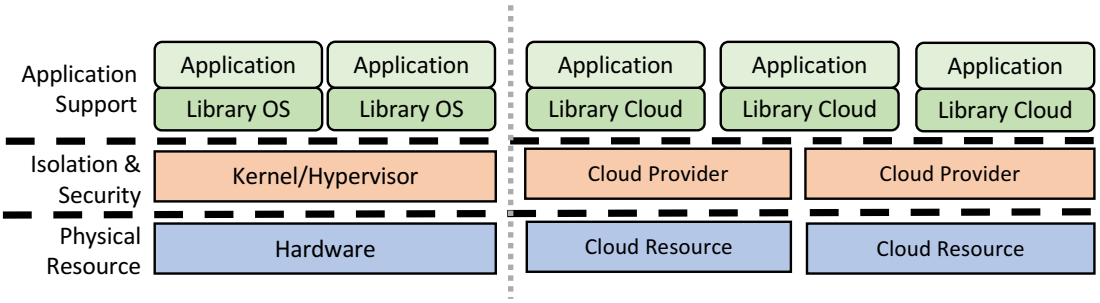


Figure 3.1: Library OS vs. Library Cloud.

geneity and implement a Library Cloud without the support of underlying cloud providers.

- We propose storage and networking solutions for supporting efficient VM migration in a Library Cloud, and show how they can be implemented efficiently.
- We evaluate a prototype of the Library Cloud abstraction—the Supercloud.
- We demonstrate how applications can benefit from the Library Cloud using case studies of geographically shifting workloads and virtual machine consolidation.

3.2 Towards the Library Cloud

3.2.1 The Library Cloud Abstraction

A *Library OS* [57, 100, 89] is an application of the end-to-end principle [104] to operating systems. A traditional monolithic OS kernel packs many functions into kernel space and presents complex high level abstractions to user programs such as a hierarchical multi-user file system and network sockets. Al-

though such high-level abstractions simplify development of applications, they also limit customization and optimization as they force applications to build on top of modules designed with specific requirements in mind. The Library OS concept, in conjunction with an *Exokernel*, addresses this problem. The Exokernel is a tiny kernel that implements only resource multiplexing and security isolation; other traditional OS functionalities are implemented in Library OSs specialized for different applications. This approach gives applications more flexibility to choose a proper abstraction for optimal performance and security.

The Library Cloud abstraction introduced in this chapter is conceptually similar to a Library OS: users and applications get control over a full cloud software stack and can customize it for their own purposes; the underlying cloud providers only need to provide basic resource multiplexing and isolation. The Library Cloud abstraction further extends the notion of Library OS: a single Library Cloud can span across multiple cloud providers. Figure 3.1 compares the Library OS and Library Cloud designs.

Applications running in a Library Cloud interact using a *Library Cloud API* that can be much richer than a typical cloud API. A Library Cloud API not only supports typical Cloud API methods such as creating VMs and managing networks, but also enables methods that are usually only available to cloud providers, such as live VM migration, consolidation, checkpointing, and dynamic scaling. Section 3.3 presents an example of the Library Cloud API. Figure 3.2 illustrates the difference between traditional clouds and Library Clouds.

A Library Cloud can be implemented on top of any existing cloud API. We are expecting that Cloud APIs will emerge with lower abstractions so that unnecessary abstraction layers can be removed. For example, HIL [71] is an

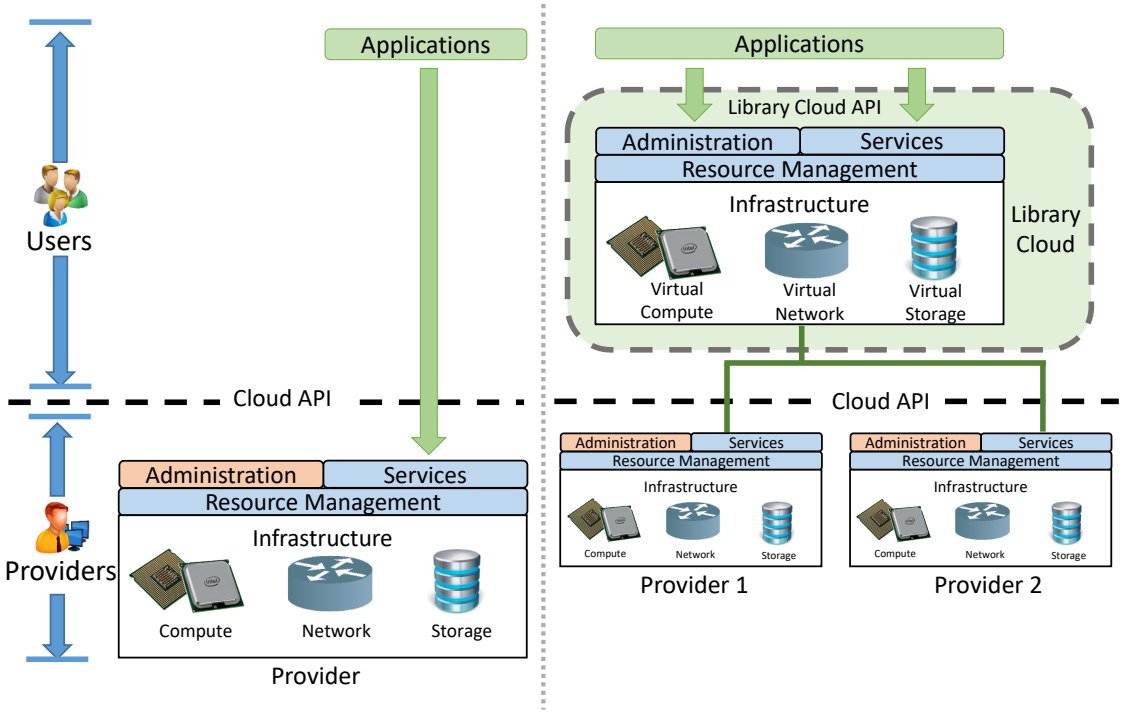


Figure 3.2: Traditional Cloud (left) vs. Library Cloud.

Exokernel-like layer for data centers that exposes physical resources directly to different cloud users.

3.2.2 Innovations Enabled by the Library Cloud

A Library Cloud abstraction supports innovations that are hard to realize in current cloud infrastructures. In this section we will describe use cases that benefit from a Library Cloud like the Supercloud.

Follow the Sun

For a service that has global users in different timezones, it would be ideal if it could "follow the sun," that is, continuously shifting to a location where the majority of users experience the lowest possible latency. This is a challenging

task even for distributed applications that can migrate by adding and removing nodes, not to mention legacy applications that cannot be easily scaled dynamically.

Distributed applications typically adopt a distributed consensus or transaction protocol in order to support data replication, distributed locking, distributed transactions, and so on. These protocols are designed to be fault-tolerant. However, adding and removing nodes while tolerating failure is a fundamental and challenging problem—it requires changing the "membership" of the system, thus subsequent requests following the membership change must be processed with a different configuration. In addition to the complexity of reaching agreement on configuration, adding and removing nodes triggers complicated internal state transfer and synchronization protocols, and membership reconfiguration is not transparent to clients. As a result, tolerating failure while providing good performance during reconfiguration is non-trivial.

Because membership reconfiguration is generally considered a rare event, applications use ad hoc mechanisms to achieve it with unpredictable performance. For example, MongoDB does not allow changing the sharding key on the fly, and the whole database must be exported and then imported again to change key distribution. As another example, re-configuring a cluster running an old version of ZooKeeper (without the new dynamic reconfiguration feature) requires a "rolling restart"—a procedure whereby servers are shutdown and restarted in a particular order so that any quorum of currently running servers includes at least one server with the latest state [109]. If not performed correctly, one server with outdated state might be elected as the new leader and the whole cluster might enter an inconsistent state. While ZooKeeper recently added sup-

port for dynamic reconfiguration, it is inefficient for geographic migration (see Section 3.4.2). A key-value store such as Cassandra can easily add and remove a node and adjust token distribution, but doing these for geographic server migration triggers unnecessary data replication and load re-balancing, and can affect service availability if a failure occurs at the same time (see Section 3.4.2).

In order to "follow the sun", applications must be migrated several times a day, and, as we have discussed, implementing migration by adding and removing nodes is suboptimal. An alternative approach is to use live VM migration since it can be performed transparently to the application and even to clients. For example, if live VM migration were supported, we can migrate the ZooKeeper leader and servers to locations where the leader and a majority of servers are geographically close to most active users, without the need of changing a single line of source code in ZooKeeper. Often this only involves migrating the leader: If the majority of clients reside in two different regions like the US and Asia, and there are $2f + 1$ servers, including one leader, then with f servers running in one location, the US, and f servers running in another, Asia, only the leader needs to migrate back and forth once a day. Live migrating a VM comprises multiple phases [51]. First, the image is copied while the VM is still running at the old location. Pages that are written by the running VM after they have been copied have to be copied again, and thus this phase might require multiple rounds of copying. When the number of dirty pages is below some minimum, the VM is suspended for a short period of time to finish copying the remaining state. After this, the VM at the new location resumes execution. The application downtime corresponds to this relatively short final copying phase plus the time required for the network layer to adjust routing paths for the migrated VM. The downtime due to live migration of a VM be-

tween the U.S. and Asia is less than one second, while the total migration time for 1GB memory (which proceeds in the background) is about 100 seconds. The downtime is small enough such that TCP connections do not break nor cause broken sessions or leader re-election.

Dynamic Resource Scheduling

VMware vSphere Distributed Resource Scheduling (DRS) redistributes, invisible to applications, virtual machines among a pool of hardware servers to optimize utilization, and it is an important technique to meet business goals. Public clouds provide services such as Amazon Auto Scaling [1] to dynamically add and remove nodes for distributed applications. But these services require users to reconfigure their application on the fly to deal with a changing number of VMs. The Supercloud supports a new paradigm for resource scaling that we call *Supercloud Dynamic Resource Scheduling* (SDRS). SDRS opens up new opportunities to minimize cost without compromising application performance or changing application configurations.

SDRS leverages hypervisor-level mechanisms enabled by the Supercloud such as VM migration, CPU capping [107], memory ballooning [119], page sharing [68], and I/O throttling [81] to share resources efficiently, while ensuring that different VMs do not interfere with each other. When load increases, nested VMs are migrated to separate provider VMs with more available resources in order to maintain application performance. This is analogous to a cloud provider consolidating VMs on a smaller number of physical machines and migrating VMs to an increased number of physical machines when load increases. When application load is low, nested VMs can be consolidated on a single provider VM, greatly reducing cost. Section 3.4.3 evaluates this using the TPC-W bench-

mark.

Smart Spot Instances

Amazon EC2 provides a spot market, offering cheap VMs in underused availability zones. Microsoft Azure has a similar offering. Such instances are difficult to use effectively in practice: prices of spot market instances change rapidly and can even significantly exceed on-demand VM instances. More problematically, Amazon EC2 reserves the right to terminate instances when an availability zone is no longer underutilized. We have designed and built *Smart Spot Instances* [76], achieving both low cost and high availability by migrating VMs rapidly to the cheapest and most available locations. The technique takes into account the network charge for transferring VM images across availability zones. In our experience, we are able to provide a long-running smart VM instances at approximately a third of the price of a normal instance. A similar idea was developed in SpotCheck [106]. The Supercloud makes it easy to develop such innovations and, going beyond SpotCheck, generalizes to multiple cloud providers.

3.3 The Supercloud: A Library Cloud Implementation

The Supercloud is an instance of a Library Cloud that is designed to operate without any specific support from underlying cloud providers. It embodies control of compute, networking, storage, and management that can be customized by users (Figure 3.2). Importantly, a Supercloud can span multiple availability zones of the same provider as well as availability zones of multiple cloud providers and private clusters (see Figure 3.3). To accomplish this, there are two layers of hardware virtualization. The bottom layer, called *first-layer*, is the in-

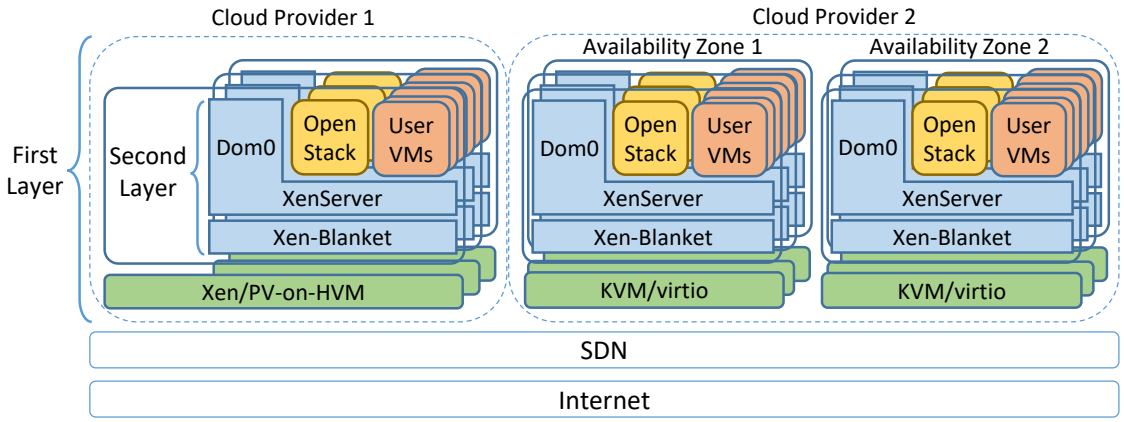


Figure 3.3: Example deployment of the Supercloud.

frastructure managed by a Infrastructure as a Service (IaaS) cloud provider such as Amazon EC2 or Rackspace, or managed privately. It provides VMs, cloud storage, and networking. Another layer of virtualization on top of this, called *second-layer*, is the IaaS infrastructure managed by the Supercloud. It leverages resources from the first-layer and provides a single uniform virtual cloud interface. Importantly, the second-layer is completely controlled by users.

In case of compute resources, the first-layer has a hypervisor managed by the underlying cloud provider and a collection of hardware virtual machines (HVMs). We refer to these as *first-layer hypervisors* and *first-layer VMs*.

The second layer, exposed to Supercloud users, is similarly separated into a hypervisor and some number of guest VMs that we call the *second-layer hypervisor* and *VMs*[1]. We use Xen-Blanket [122] for the second-layer hypervisor. Xen-Blanket provides a consistent Xen-based para-virtualized (PV) interface. In Xen, one VM is called *Domain-0* (aka *Dom0*) and manages the other VMs, called *Domain-U*s (aka *DomU*s). A second-layer Dom0 multiplexes resources such as I/O amongst the DomUs.

[1]User VMs, user VM instances, second-layer VMs, and second-layer DomU VMs are used interchangeably.

We use OpenStack [16] to manage user VMs and provide the administrative OpenStack API as the Library Cloud API to existing applications. In particular, a XenServer runs within the second-layer Dom0 and allows OpenStack to manage all second-layer DomU VMs.

A common practice in VM migration is using shared storage to serve VM images, so that a VM migration only needs to transfer the memory. This is essential to good performance because migrating a VM with the disk image takes a long time. XenServer offers two options for sharing storage: an NFS-based solution and an iSCSI-based solution. Both these approaches use a centralized storage repository. While simple, this can lead to significant latencies, low bandwidth, and high Internet cost for VMs that access the disk through a wide area network when migrated to another region or cloud. Previous works have proposed different mechanisms and optimization for wide-area VM migration with disk images [45, 90, 96]. However, migrating the whole image file incurs significant Internet traffic, which is typically charged by cloud providers. To support efficient VM migration in the wide area network, we developed a geo-replicated image file storage that seeks a good balance between performance and cost.

To provide the illusion of a single virtual cloud, the control services (including XenServer and OpenStack) and user VMs need to communicate in a consistent manner no matter where the end-point VMs reside. A migrated user VM expects its IP address to remain unchanged. To accomplish this, the Supercloud network layer is built using a Software-Defined Network (SDN) overlay based on Open vSwitch, VXLAN tunnels, and the Frenetic SDN controller [59]. Such an overlay network gives control over routing for the second-layer and enables compatibility with heterogeneous first-layer networks.

We support all major hypervisors including Xen, KVM, Hyper-V, and VMware, so a Supercloud instance can span all major cloud providers including Amazon EC2, Rackspace, Windows Azure, Google Compute Engine, and VMware vCloud Air.

The Supercloud, like any other IaaS cloud, provides three types of resources: computing, networking, and storage. Below we present each one in more detail.

3.3.1 Computing

Nested Virtualization

Nested virtualization is essential in the Supercloud in order to provide a uniform interface and support privileged operations such as live VM migration. However, nested virtualization is not natively supported by most cloud providers. Solutions that require special support in the first-layer hypervisor such as the Turtles project [38] cannot be run in public clouds. We use XenBlanket [122], a nested virtualized Xen hypervisor that runs on various widely supported hardware virtualized VMs (HVMs) and provides a para-virtualized (PV) interface to second-layer VMs.

In a full virtualized environment, the underlying hypervisor is completely transparent to the system running in the VM. When virtual devices are emulated, a Xen hypervisor can run without modification in an HVM. However, emulated devices have poor performance. Thus, clouds typically adopt para-virtualized (PV) devices for HVM instances, e.g., `PV-on-HVM` devices on Xen, `virtio` devices on KVM, and `enlightened I/O` devices on Hyper-V. In this model, a device driver in the VM works together with the underlying hypervi-

sor to improve I/O performance. This solution breaks transparency, and brings new challenges when we want to run Xen as a second-layer hypervisor:

- A standard PV device driver cannot work properly because a second-layer Domain-0 (Dom0) is no longer running in the `Ring-0` privileged domain;
- The notion of a "physical address" in a Xen-based VM no longer matches the "machine address" in the HVM.

Xen-Blanket opens new interfaces to enable communication between a PV device driver in the second-layer Dom0 and the first-layer hypervisor, and leverages different *blanket drivers* to deal with diverse hypervisors. It then multiplexes these PV devices and provides a consistent Xen device interface. Hence the underlying infrastructure becomes transparent to the second-layer user VMs. The bulk of the effort in enabling Xen-Blanket in a cloud infrastructure is porting the blanket driver. Once enabled, second-layer user VMs can run on Xen-Blanket without any modification.

The performance overhead of Xen-Blanket has been thoroughly evaluated [122]. To summarize, Xen-Blanket is able to match native network I/O performance and incur about 12% overhead for disk I/O performance compared to a native para-virtualized instance. We also tested a sysbench CPU benchmark, which shows that the completion time difference between first and second layer VMs is within 10%.

Handling Heterogeneity

Different underlying CPUs might have different capabilities, preventing a VM from working properly after being migrated. We can record the CPU features

exposed to a second-layer VM when it is created, and only migrate it to a first-layer VM that support compatible CPU features. However, this places a limit on the placement of the VM.

For applications that require more flexibility on VM placement, we can expose a subset of CPU features that are common in CPUs, which is sufficient for supporting most applications. This requires a solution to control which CPU feature we want to expose. Hardware solutions such as Intel VT FlexMigration and AMD-V Extended Migration trigger a fault when guest VMs execute a `CPUID` instruction, thus providing the virtual machine monitor (VMM) an opportunity to intercept this instruction and report a consistent set of CPU features to the VMs. Unfortunately, `CPUID` faulting is not virtualized. As a result, a second-layer hypervisor running in an HVM loses the ability to intercept the `CPUID` instruction.

Fortunately, because all second-layer VMs running on top of Xen-Blanket are para-virtualized, in most cases they will invoke the `pv_cpuid` hook in the hypervisor instead of executing the `CPUID` instruction directly. So we modified the `pv_cpuid` hook in Xen-Blanket to only expose a desired subset of CPU flags. Applications running in second-layer VMs will see a set of consistent CPU flags when checking the Linux */proc* interface. Note that this solution does not prevent user VMs from running the `CPUID` instruction directly. Processor vendors may eventually support virtualized `CPUID` faulting so that a second-layer hypervisor could intercept the instruction and control CPU flags exposed to second-layer VMs.

3.3.2 Storage

Consistency and Data Propagation

Geo-replicated storage solutions sometimes adopt a weaker consistency model, such as *eventual consistency*, in which applications reading different replicas may see stale results. This is not suitable for an image storage on which a migrated VM expects up-to-date data. Using a strongly consistent geo-replicated storage requires synchronous data propagation, resulting in low write throughput. A key observation is that a running application typically does not require all data in the image. Our storage system thus decouples consistency from data propagation.

As shown in Figure 3.4, the storage service consists of two layers: a data view layer provides a required consistency guarantee, and a data store layer stores and propagates data. Images are divided into blocks with a constant size (4KB). The global meta-data in the data view layer includes a version number for each block. When a VM is reading a block, the version number is compared to the version number in the local meta-data in the local data store. If the latest version of the block is available locally, data is returned immediately. Otherwise, the data store checks the location of the block in the global meta-data and fetches the data remotely. Updating a block increases its version number by one, and the updated global meta-data is then propagated to other replicas.

The consistency model seen by applications is completely determined by the data view layer. Since we only have global meta-data in this layer, it is relatively cheap to implement strong consistency. Section 3.3.2 describes how

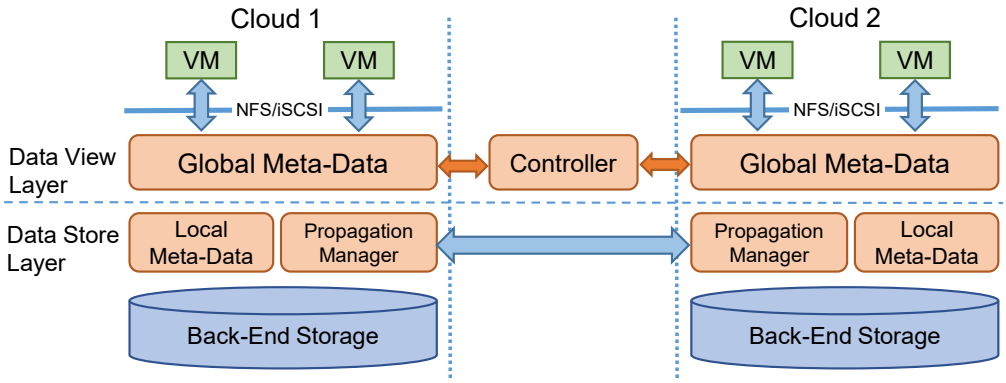


Figure 3.4: Storage architecture of the Supercloud.

we can further optimize the synchronization of global meta-data by relaxing the consistency model. Since the data store layer is decoupled, data propagation can be optimized separately without concern about consistency.

Our current design optimizes performance and minimizes traffic cost. The Back-End Storage can tolerate failures within a single cloud. However, after several migrations the image may have the latest version of blocks scattered in different clouds, and a cloud-level failure before an updated block is propagated may affect the availability of the whole image. If an image is really critical and would like to tolerate cloud failures, it is possible to pass a hint to the propagation manager so that each write is synchronously propagated to different clouds, at significant cost to application performance.

Global Meta-Data Propagation

Due to the long latency in the wide area network, meta-data transfer affects application performance significantly if this propagation is in the read/write critical path. We make two key observations:

- If an image file is open for writing, it can only be accessed by a single VM.

This happens when the image is private to a VM.

- If an image file is shared by multiple VMs, it is read-only. This happens when the image is a snapshot or a base image file.

These observations indicate that, although a migrated VM is expecting strong consistency from the image storage, multiple replicas of the same image file do not need to be identical all the time. It suffices to provide VMs with a "close-to-open" consistency: after a file is closed, reads after subsequent opens will see the latest version. We remove the global meta-data propagation from the read/write critical path by committing the meta-data update locally, and only flushing the global meta-data to a centralized controller when closing the image file. Subsequent opens of the file need to sync the global meta-data with the controller first. Note that the controller is involved only when opening or closing the image file.

Data Propagation Policies

By decoupling the data view and data store layer, a VM can see a consistent disk image no matter where it is migrated. However, fetching each block on-demand through the wide area network significantly degrades read performance. It is useful to proactively propagate a block before migration if we can predict that it is going to be accessed in another place. Intuitively, a block that is read frequently and updated rarely should be aggressively propagated. On the other hand, propagating a block that is updated frequently is a waste of network resources. Because Internet traffic is typically charged to the user, we need to be careful about proactive propagation.

The data store layer monitors the access pattern of the VM and selectively

propagates those blocks that are most likely to be accessed after migration and least likely to be updated after propagation. To this end, we maintain a priority queue of updated blocks for each image file. The priority of each block is updated on each read or write. We use the read/write ratio $F = f_r/f_w$ to calculate the priority of propagating a block, where f_r and f_w are the read and write frequency of a block respectively. When two blocks have the same value of F, we first propagate the block with a higher read frequency. So the formula of calculating the propagation priority P^b for a block b is:

$$P^b = \begin{cases} K \cdot f_r^b / f_w^b + f_r^b & \text{if } f_r^b / f_w^b \geq S \\ -1 & \text{if } f_w^b = 0 \text{ or } f_r^b / f_w^b < S \end{cases}$$

The value K determines how much we want to favor the read/write ratio. In our prototype we use $K = 1000$. The constant S determines the aggressiveness of the propagation. The lower S is, the more data will be propagated. When $f_w^b = 0$, which means the block has never been updated, or when $f_r^b / f_w^b < S$, we set the priority to -1 to indicate that this block is not to be propagated proactively. In our current prototype, S is set to 1.

Applications might have different requirements for when to propagate data updates. Depending on the workload or the trade-off between application performance and traffic cost, the propagation should be tuned separately for each VM on the fly. Our storage layer provides parameters K and S as tuning knobs for this purpose. Designing an automatic online tuning policy is left as future work.

After deciding which block to propagate, the next question is where to propagate the block. Since the Supercloud can be deployed to many places even

across clouds, propagating each block to all destinations wastes Internet traffic and money. To further optimize data propagation, a transition probability table is added into the image file's meta-data, indicating the probability that a VM is moved from one place to another. Each data block is randomly propagated according to the probability in the transition table, so that the destination to which the VM is most likely to be migrated will receive most propagated blocks. In our current prototype, the transition table is provided as a hint by the user when creating the image file. It is also possible to train the table on the fly if the VM is migrated many times.

3.3.3 Networking

High-Performance VPN

To enable communication between control services (including XenServer and OpenStack services) and VMs, we place them into a virtual private network (VPN). Good performance requires minimizing the number of hops. Existing VPN solutions such as OpenVPN [17] use a centralized server to forward traffic, which causes high latency and poor throughput. The *tinc* VPN [23] implements an automatic full mesh peer-to-peer routing protocol, minimizing the number of hops traversed between endpoints. However, we found that tinc imposes high performance overhead, mostly caused by extra data-copy and kernel/user mode switching.

To build a high-performance VPN solution for the Supercloud, we use Open vSwitch [15], VXLAN tunnels, and the Frenetic SDN controller [59]. Open vSwitch implements data-paths in kernel mode and supports an OpenFlow-

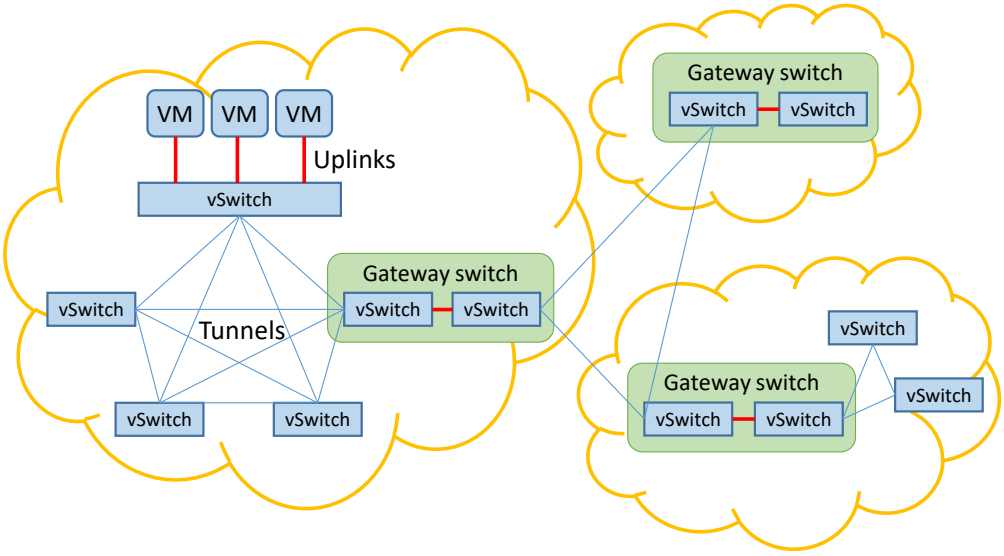


Figure 3.5: Network topology of the Supercloud.

based control plane. Each virtual switch uses an uplink to connect to the VMs running on the same first-layer VM and a set of VXLAN tunnels to connect to all other switches, as illustrated in Figure 3.5. We create a full mesh network here because we want to always forward packets directly to their destinations. We use VXLAN over GRE (Generic Routing Encapsulation) tunnels because this approach is based on UDP instead of a proprietary protocol, and thus better supported by different firewalls.

In a hierarchical topology, switches running in a private network cannot set up VXLAN tunnels directly with other switches outside the network. A gateway switch is required to forward packets in this case (see Figure 3.5). The node running the gateway switch is in more than one network. To implement the gateway switch, we create one switch for each of the networks, inter-connected with an in-kernel *patch port*. Each switch builds full mesh connections with other switches in its own network as before and treats the patch port as an uplink.

Switches connected in a full mesh form loops. Ordinarily one would run

a spanning tree protocol, but with a network topology demonstrated in Figure 3.5, a spanning tree cannot minimize the number of hops for every pair of nodes. To route packets efficiently, switches in the Supercloud VPN are connected to a centralized SDN controller implemented with Frenetic [59]. The controller learns the topology of the network by instructing the switches to send a "spoof packet." On receiving the spoof packet, switches report to the controller on which port the packet is received, so that the controller can record how switches are connected. The controller implements a MAC-learning functionality for each switch. The MAC address, IP address, and location of a VM is learned when it sends out packets.

ARP (Address Resolution Protocol) packets are forwarded directly to the destination instead of broadcast. When routing a packet, the controller calculates the shortest path on the network and installs OpenFlow rules along the path to enable the communication. This is done only once for each flow and subsequent data transportation does not need to involve the controller anymore.

Supporting VM Live-Migration

Supporting VM live migration is another challenge. To keep the IP address of a migrated VM unchanged, the underlying VPN needs to adjust the routing path and re-direct traffic. However, the adjustment cannot be done immediately when the migration is triggered, because at this point the VM is still running in the original location and existing network flows should not be affected.

In order to know at which point the routing path should be adjusted without adding a hook into the hypervisor, before triggering the migration, the controller is notified with the source and destination of the migrated VM. The con-

troller then injects a preparation rule in the destination switch so that it can get an immediate report when a packet with the migrated VM's MAC address is received. When migration is finished, the migrated VM will send out an ARP notification. This is captured by the controller so that it knows that the migration has finished. The controller then updates all switches that have the migrated VM's MAC address in the MAC table, avoiding the usual ARP broadcast.

Supporting Public IP Addresses

A public IP address is required to expose a service to the public network. We currently support addressing a second-layer VM from the public network by allocating a public IP address to a first-layer VM in the Supercloud network (i.e., a public IP front-end), and then applying port forwarding to map certain ports to the second-layer VM. This solution has good performance because packets can be routed in the public network to the VM directly.

A challenge arises when this VM is migrated to a different cloud provider. Without specific support from Internet Service Providers (ISPs) such as anycast, mobile IP, or multihoming, traffic sent to the public IP address needs to be re-directed by the same first-layer VM, no matter where the second-layer VM currently resides. While this works, it can lead to high latency.

To address this issue, we adopt an idea from Content Distribution Networks (CDN): instead of giving the same public IP address to clients all over the world, we give each client the public IP address of a front-end server in a nearby data center. Taking this extra but short hop, the routing from the front-end to the second-layer VM is always optimized in the public network, performing much better for most clients than a centralized public IP solution.

For applications that do not want to pay the additional cost of the front-end servers, multiple public IP addresses, and an additional hop, the Supercloud has its own dynamic DNS service and can update the DNS mapping after migration. Note that clients might see an out-of-date public IP address before the DNS cache is expired. For services that use protocols such as HTTP, SOAP, and REST, we deploy an HTTP redirection service on the original public IP front-end and respond to clients with an HTTP redirection response.

3.3.4 Management and Scheduling Framework

A Library Cloud includes a resource management platform that manages underlying computation, storage, and network resources and that provides an easy-to-use Library Cloud API to access them. To this end, we adopted the widely used OpenStack. OpenStack provides interfaces to manage cloud resources through a web-based *OpenStack dashboard*, or via the *OpenStack API*. To incorporate OpenStack with Xen-Blanket, we run XenServer [24], a server virtualization platform that provides feature-rich APIs to communicate with the Xen hypervisor.

One technical challenge that we faced was that XenServer requires a direct installation with an ISO image of the complete software stack, including the Xen hypervisor and Dom0 operating system (OS), which cannot be easily ported to different clouds. To overcome this, we use XenServer-core [25], a set of core components of XenServer that can be installed in a standard CentOS installation. We ported Xen-Blanket to Xen 4.2.2 and Linux Kernel 3.4.53 in order to run XenServer-core.

Applications in the Supercloud can make migration decisions by themselves

and issue migration commands through the OpenStack API. To facilitate the process of deciding optimal placement, the Supercloud provides a scheduling framework for the user. Users can customize the scheduling policy for different applications by implementing some interfaces. The Supercloud scheduler periodically evaluates current placement of the application and automatically adjusts it when a better placement is found.

Suppose the Supercloud is deployed to N different data centers: $d_1, d_2, ..., d_N$. We denote a placement plan for an application with k nodes as $P = \{p_1, p_2, ..., p_k\}$, where p_i is the location of the i^{th} node. Periodically, the Supercloud measures end-to-end latency between all different data centers and stores the results in a latency matrix L, where $L(i, j)$ is the round-trip-time (RTT) from d_i to d_j. The workload of the application is also captured periodically in a workload statistics report S. S is application-specific, so it should be monitored in the application-level and passed to the scheduling framework as an opaque handle. Applications can pack more information into S if needed, such as the current placement, topology, and workload history.

In order to evaluate a placement plan, the application has to provide 1) an evaluation function $f(P, S, L)$ that evaluates a placement plan under a certain workload and returns a score, and 2) a threshold T that specifies the minimal score change that can trigger VM migration. Using f, the scheduler iterates through all possible placement plans and gets a set of candidate placement plans D that maximize the score and outperform the current placement plan $P_{current}$ by at least T, that is:

$$D = \arg\max_{P} \{f(P, S, L) | f(P, S, L) \geq f(P_{current}, S, L) + T\}$$

To choose a placement plan in D, the scheduler compares each placement plan with the current placement $P_{current}$ and selects the one that requires the fewest migrations.

Below we present case studies to demonstrate policies for two different types of applications.

Non-Distributed Applications

For a single VM running a non-distributed application such as a MySQL database, a placement plan is simply the location of the VM: $P = \{p\}$. We can deploy a set of service front-ends located in different data centers to collect user requests and forward them to the VM. This architecture makes the location of the VM transparent to clients and can help with simplifying placement evaluation (Section 3.3.3).

The goal of placement is to minimize average latency for all front-ends. Here we only consider latency in the network and ignore processing time for different types of requests. S is defined as:

$$S = \{(s_1, d_{s_1}), (s_2, d_{s_2}), ..., (s_n, d_{s_n})\}$$

where n is the number of front-ends, s_i is the number of active clients of the i^{th} front-end, and d_{s_i} is its location.

Each front-end is assigned a weight, which equals the number of requests it receives. The score of a placement plan is the weighted average latency of all front-ends (negated so lower latencies result in higher scores), that is,

$$f(P,S,L) = -\sum_{i=1}^{n} s_i \cdot L(d_{s_i}, p)$$

Distributed Applications with Replicated State

Distributed applications typically maintain replicated state using some consensus protocol. Below we use ZooKeeper as an example. ZooKeeper is implemented using a replicated *ensemble* of servers kept consistent using Zab (ZooKeeper Atomic Broadcast) [77]. In Zab, one server acts as a leader that communicates with a majority of servers to agree on a total order of updates. Read requests are handled by any node in the ensemble, while write requests must be broadcast by the leader and agreed upon by the majority of the ensemble.

For a ZooKeeper ensemble with m nodes, a placement plan is the location of all nodes: $P = \{p_1, p_2, ..., p_m\}$, where p_l is the location of the leader. Again we assume that there are n front-ends in different data centers to collect and forward client requests. To evaluate a placement plan, we need to consider read and write requests separately. So:

$$S = \{(r_1, w_1, d_{s_1}), (r_2, w_2, d_{s_2}), ..., (r_n, w_n, d_{s_n})\}$$

r_i and w_i are the number of read and write requests received by the i^{th} front-end and d_{s_i} is its location.

The goal of placement is again to minimize average network latency for all front-ends, ignoring request processing time and only considering network delay. For the purpose of load balancing, ZooKeeper clients are typically con-

nected randomly to one node in the ensemble. Read requests can return immediately. For the i^{th} front-end, the expected read latency is

$$R_i = \underset{j=1...m}{\text{avg}} L(d_{s_i}, p_j)$$

Write requests are processed in three steps:

- Step 1: a write request from the i^{th} front-end goes randomly to one of the ZooKeeper nodes. The average latency is:

$$W_i^{(1)} = \underset{j=1...m}{\text{avg}} L(d_{s_i}, p_j)$$

- Step 2: the write request is then forwarded to the leader of the ensemble. The average latency for this step is:

$$W_i^{(2)} = \underset{j=1...m}{\text{avg}} L(p_j, p_l)$$

- Step 3: the leader broadcasts the request twice in a protocol similar to two-phase commit, and for each broadcast it must wait until at least half of the ensemble replies. The average latency for this step is:

$$W_i^{(3)} = 2 \times \underset{j=1...m, j \neq l}{\text{median}} L(p_l, p_j)$$

So the expected network latency for a write request from the i^{th} front-end is: $W_i = W_i^{(1)} + W_i^{(2)} + W_i^{(3)}$. The evaluation function for ZooKeeper calculates the weighted average network latency of all requests, assuming that we give read and write requests a weight α and β respectively:

$$f(P,S,L) = -\sum_{i=1}^{n}(\alpha \cdot R_i \cdot r_i + \beta \cdot W_i \cdot w_i)$$

3.3.5 Discussion

The benefits of the Supercloud do not come without costs. For example, nested virtualization imposes performance overhead including CPU scheduling delay and I/O overhead. Users can evaluate this tradeoff based on migration frequency and performance requirements, and it is application-dependent. We are currently in the process of building support for containers into the Supercloud. Container technology [110] provides another way to homogenize different cloud platforms. However, compared to live VM migration, which is mature and widely used, container migration [94] technology is preliminary and involves a checkpointing/resume mechanism that might cause a relatively large performance hiccup. Even with mature container migration, the challenges of building a well-performing Supercloud remain essentially the same.

3.4 Evaluation

The Library cloud represents a powerful abstraction and its implementation via the Supercloud represents a new and unique capability: A distributed service can migrate live, and incrementally or whole, between availability zones and heterogeneous cloud providers. In this section, we investigate four research questions enabled by and enabling this abstraction and capability:

1. How effective is the abstraction and its scheduler in enabling applications to follow-the-sun?

2. Is VM migration a viable approach to follow-the-sun?
3. How effective is SDRS in saving the cost of running an application?
4. What is the efficacy of Supercloud storage and networking in supporting live VM migration?

3.4.1 Follow the Sun

In this set of experiments, we use a distributed application, ZooKeeper (Section 3.4.1), and a database, MySQL (Section 3.4.1), to investigate application performance benefits due to following the sun. Results demonstrate that the Supercloud scheduler was able to automatically follow the sun and migrate resources geographically and across heterogeneous clouds, enabling high performance to be maintained.

A ZooKeeper Ensemble

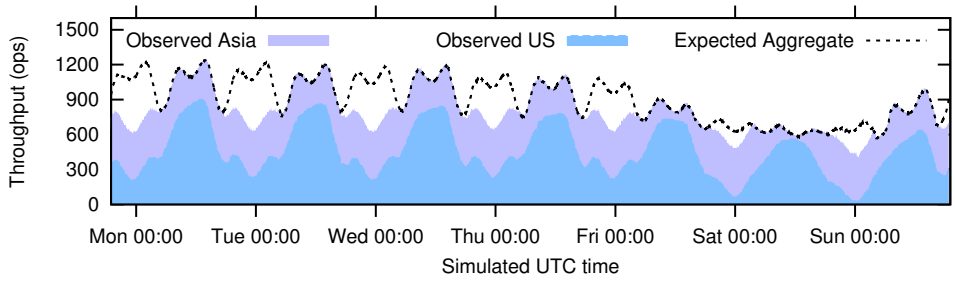
ZooKeeper writes require a majority of ZooKeeper servers (aka, the ensemble) and a ZooKeeper leader to coordinate and order all writes. High network latency between ZooKeeper servers or between clients and the ZooKeeper leader causes high end-to-end service latencies. For good performance, the majority of ZooKeeper servers have to be located where most active clients are, and the ZooKeeper leader must be part of that majority.

Experiment setup: To evaluate the efficacy of following the sun using the scheduler described in Section 3.3.4, we used a ZooKeeper distributed application and measured its ability to respond to clients in two different regions, Virginia and Taiwan. The clients used a workload corresponding to a weeklong

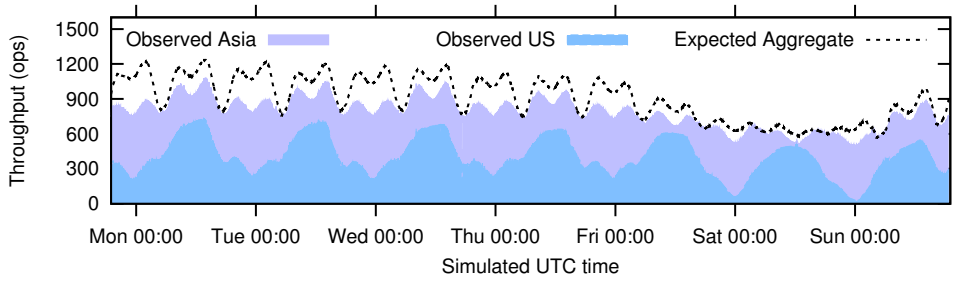
trace of active MSN connections [49, Figure 5a]. The trace does not specify the start time of the MSN workload—we made an educated guess based on the diurnal pattern in the data and assumed that the workload started at 12am at the beginning of a Monday. Nor does the trace specify the absolute workload—we scaled the workload so that the peak workload can be served by our ZooKeeper cluster when latency is low. We varied load based on location and time: We circularly shifted the start of the trace by 12 hours ahead to produce an identical workload where the start of the trace from Virginia was 12 hours before starting in Taiwan. The ZooKeeper clients randomly connected to one ZooKeeper server and submitted blocking read and write requests in a ratio of 9:1. Each read operation obtained a 64-byte ZooKeeper *znode*; each write operation overwrote a 64-byte znode. The clients ran on first-layer VMs in Virginia on Amazon VMs and in Taiwan on Google VMs.

For the ZooKeeper ensemble, we deployed the servers in the Supercloud simultaneously spread across Amazon Virginia and Google Taiwan regions. The type of first layer VMs used in our experiments was `m3.xlarge` in Amazon and `n1-standard-4` in Google, both of which had 4 vCPUs and 15GB memory. The ZooKeeper ensemble ran on three second-layer VMs, denoted as `zk1`, `zk2` and `zk3`. Each VM had 1GB RAM and 1 CPU core. We used one first layer VM in each region to serve as the Supercloud storage server. The data of the ZooKeeper nodes was stored on disk and propagated automatically by the storage servers.

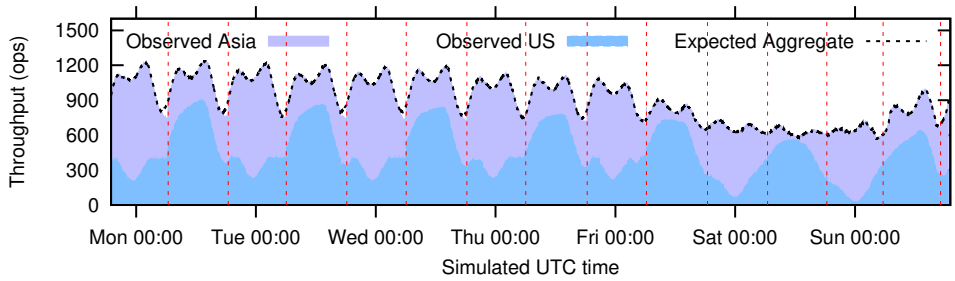
We implemented the scheduling evaluation functions for ZooKeeper discussed in Section 3.3.4. ZooKeeper server VMs reported VM load to the scheduler. How quickly the scheduler reacts to workload changes depends on the fre-



(a) US Ensemble



(b) Global Ensemble



(c) Dynamic Ensemble

Figure 3.6: ZooKeeper throughput (vertical dashed lines indicate the end of the migrations).

quency of workload monitoring and evaluation. For this experiment, scheduler placement evaluation was triggered every minute. Once a VM migration was started, the scheduler waited until it finished before considering a new placement. Migrations were performed in parallel, so going from one placement plan to another was fast.

We evaluated ZooKeeper in three scenarios:

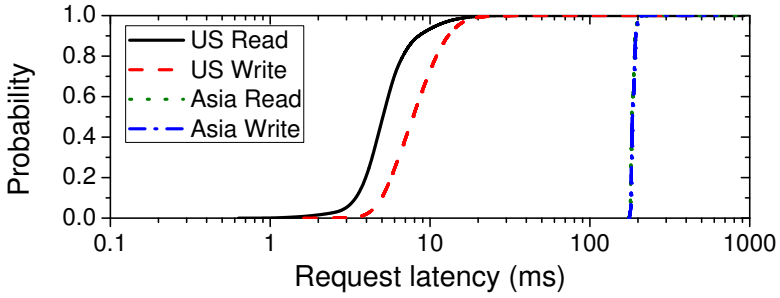
1. **US Ensemble**: all three ZooKeeper nodes, `zk1`, `zk2`, and `zk3`, were in

Amazon Virginia;

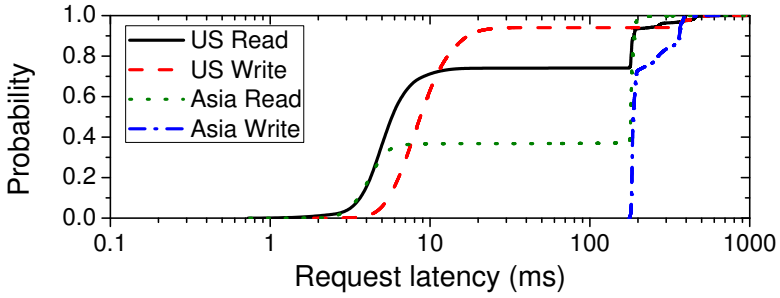
2. **Global Ensemble**: `zk1` and `zk2` were in Amazon Virginia, and `zk3` was in Google Taiwan;
3. **Dynamic Ensemble**: the Supercloud scheduler automatically placed VMs according to the workload.

To simplify experimentation and save cost, we sped up the trace by a factor of 30 so that the one week trace could be replayed in less than six hours. The performance of the US Ensemble and Global Ensemble was not affected by the speed-up, while the performance of the Supercloud was slightly degraded since migration was not sped up correspondingly.

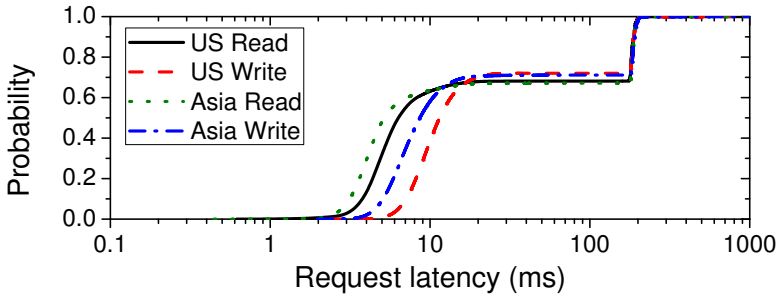
Experimental results: Figure 3.6 shows the throughput (in operations per second) for the three scenarios, US, Global, and Dynamic Ensembles. Each scenario displays throughput measured for the United States (US) clients in Virginia ("Observed US"), throughput measured for Asian clients in Taiwan ("Observed Asia"), and throughput if latency were negligible ("Expected Aggregate"). The throughput results observed by US and Asian clients are stacked on top of one another (with US below Asia), so that the top of the throughput results for the clients in Asia show the total aggregate throughput that was observed by all clients. For the US Ensemble scenario, Figure 3.6a, US clients were able to reach their maximum throughput, but clients in Asia suffered from high latencies, and, as a result, experienced poor throughput. For the Global Ensemble, Figure 3.6b, by placing one node in Taiwan, $1/3^{rd}$ of Asian clients experienced increased read throughput and total throughput was improved. However, $2/3^{rd}$ of the read requests and all write requests from Asian clients still experienced poor throughput. Moreover, US clients were not able to reach their maximum



(a) US Ensemble



(b) Global Ensemble



(c) Dynamic Ensemble

Figure 3.7: ZooKeeper latency CDF.

throughput during peak times because $1/3^{rd}$ of clients were connected to the ZooKeeper server in Taiwan. Finally, Figure 3.6c, shows the result for the Supercloud case where throughput remained high and matched the expected performance as the scheduler was able to automatically migrate ZooKeeper servers to the region where load was high.

Figure 3.7 shows the cumulative latency distributions for read and write operations. We can see that for the US Ensemble, Figure 3.7a, 90% of US clients ob-

served less than 15ms latency for read and write operations, while client latencies in Asia were close to 200ms. (We repeated the experiments with an Asia Ensemble and observed symmetric results.) In the Global Ensemble, Figure 3.7b, the ZooKeeper quorum was in the US. The server in Taiwan helped Asian clients gain better read throughput, though write latency did not improve: 37% reads from Asian clients completed in 15ms because they were handled by the server in Taiwan. The high tail latency of US read/write performance was because some requests went to the server in Taiwan. (A symmetric experiment with the quorum in Asia observed the same results.)

Finally, in the Dynamic Ensemble, Figure 3.7c, the scheduler automatically figured out that placing all three nodes in one location was the best placement for the workload. This is because clients connect to servers randomly and the read/write ratio is fixed. For example, leaving one node in Asia can only improve performance of one third of read requests from Asia, but also make one third of read requests from the US suffer from long latency. This was not beneficial when the US workload was higher than the Asia workload. Therefore, it always migrated the whole ensemble together: The scheduler migrated the ensemble every 12 hours between Amazon Virginia and Google Taiwan to make the ZooKeeper cluster close to most clients. The scheduler placement resulted in low latency: 61% of writes and 69% of reads were less than 15ms. About 30% of requests had longer latency since they were from clients accessing the servers remotely. However, on average the Dynamic Ensemble experiment achieved significantly lower and more balanced latency across all clients than US Ensemble and Global Ensemble.

A Single MySQL Database

Experiment setup: We used the `TPC-W` benchmark to stress a single MySQL database, and deployed web servers in two first-layer VMs in the Amazon Virginia and Google Taiwan regions. The TPC-W benchmark clients performed 80% browsing (read) and 20% ordering (write).

The MySQL database ran in the Supercloud in a second layer VM that had 3 vCPUs and 2GB memory. The first-layer VM could run in the Amazon Virgina or Google Taiwan regions with instances described in Section 3.4.1. The storage engine of MySQL was set to `MEMORY`.

For comparison, we evaluated three scenarios:

1. **First Layer Database**: The database was deployed in a first-layer VM located in Taiwan;
2. **Second Layer Database without Migration**: The database was deployed in a second-layer VM located in the Taiwan;
3. **Second Layer Database with Migration**: The database was deployed in a second-layer VM, and the Supercloud scheduler automatically migrated the VM back and forth between Amazon Virginia and Google Taiwan regions according to the workload.

Experimental results: Figure 3.8 shows the results: Cumulative latency distributions for web interactions. There are a several points to observe. First, for the first- or second-layer databases without migration, 80% of the clients in Taiwan ("Asian Clients") observed less than 10ms latency, while latencies for clients in Virginia ("US Clients") were close to 200ms. Comparing with the latency of

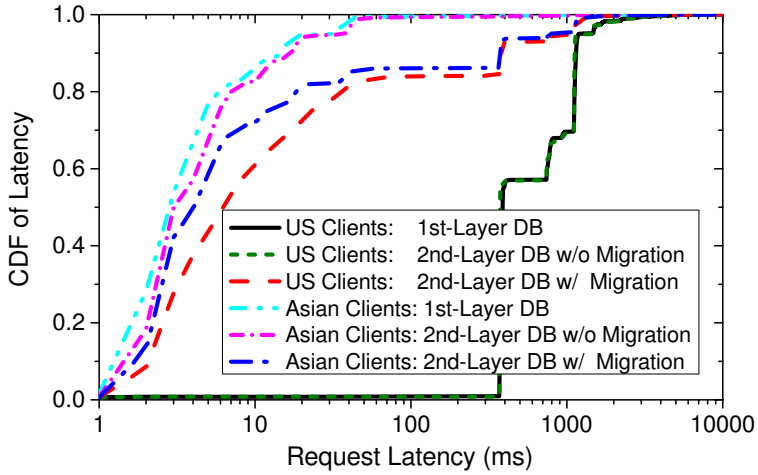


Figure 3.8: TPC-W Client Latency CDF

Asian clients in the first two scenarios, we can see a slight difference in the curve caused by the small overhead that the Supercloud presents.

In the third case, "2nd=Layer DB w/ Migration", the Supercloud scheduler automatically migrated the MySQL database every 12 hours between Taiwan and Virginia to place the database where load was high. As a result, the aggregate throughput to match the workload. Figure 3.8 shows that latencies for 70% web interactions were less than 10ms. The tail latency was caused by requests issued in the night of the corresponding region when the database was far away. The small plateaus in the curves were caused by the 20% ordering requests that write to the database and incur a higher latency. And the Asian clients observed a slightly shorter latency, which indicated that the Google VMs experienced lower latencies than the Amazon VMs.

3.4.2 Comparing Migration Approaches

In this set of experiments, we use two popular distributed applications, Cassandra and ZooKeeper, to investigate the viability of relying on live VM migration

via the Supercloud to enable distributed applications to follow-the-sun. The results demonstrate that not only can distributed applications benefit from the Supercloud, but they can also do so without any change to the application and can outperform other approaches that require application modification.

Cassandra Migration

Cassandra [5] supports adding and removing nodes and automatically handles data replication and load distribution. When following the sun, it makes little sense to move only some of the Cassandra nodes. Migrating an entire Cassandra cluster can be implemented by adding nodes in the destination location and removing nodes in the source location.

Experiment setup: To compare migration approaches, "application" and "Supercloud", we started a 3-node Cassandra cluster in the Amazon Virginia region, then migrated the whole cluster to the Google Taiwan region. As a result, the migration was across geographically separated clouds. We deployed a 3-node Cassandra cluster with replication factor of 2 in second-layer VMs with 1vCPU and 2GB memory. For the first-layer VMs, we used `m3.xlarge` instances in the Amazon Virginia region and `n1-standard-4` instances in the Google Taiwan region.

For the application migration approach, a Cassandra cluster was initially running in Virginia. To move all nodes to Taiwan, we followed the process specified in the Cassandra documentation for replacing running nodes in the cluster. Three new nodes in Taiwan joined the cluster with a configuration file pointing to a seed node and automatically propagated their information through Cassandra's gossip protocol. The key space then spread evenly across all six nodes

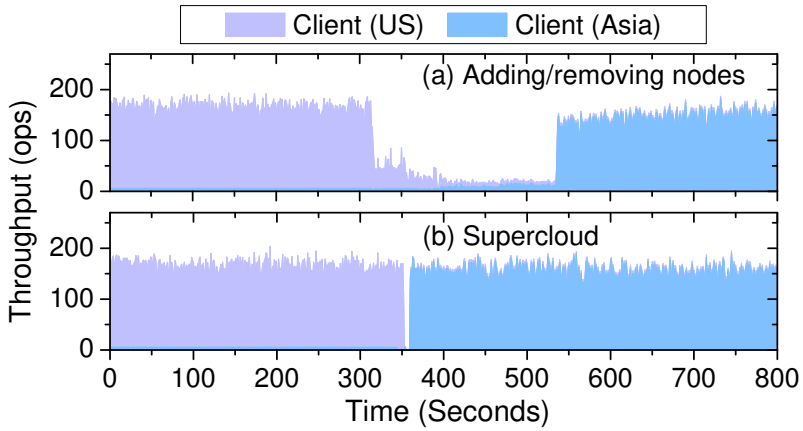


Figure 3.9: Comparison of different migration mechanisms for moving a Cassandra cluster.

and the data associated with the key moved automatically. After the three new nodes joined the cluster, we performed "node decommissioning" in each of the three original nodes. As a result, the Virginia nodes copied their data to the Taiwan nodes. The gossip protocol automatically updated each node with the cluster information transparently to the clients.

For the Supercloud migration, we set up a Supercloud across the Amazon Virginia and Google Taiwan regions using the same first- and second-layer VM configurations described above.

For the workload, data was stored in memory and moved explicitly with the application approach or migrated with the VM with the Supercloud approach. We populated the database with 30,000 key/value pairs, each of which had a size of 1KB. We started one client in each region. The clients read and wrote to the database continuously with a ratio of 4:1; each read operation obtained the value of a random key, and each write request updated the value of a random key. The consistency level was set to `ONE` (default), indicating eventual consistency.

Experimental results: Figure 3.9(a) shows the throughput of application migration (in operations per second) for both Amazon Virginia and Google Taiwan clients. Although not easy to see in this case, the throughput results for both types of clients are again stacked on top of one another, with the throughput of the Asia clients below that of the throughput of the US clients. The top of the graph therefore shows the total aggregate throughput. During migration, the throughput dropped dramatically and remained low for about 200 seconds. Even after the migration completed, it took another 200 seconds to restore performance to the original throughput due to the overhead of data replication.

With the VM migration mechanism in the Supercloud, we migrated the three Cassandra VMs from Amazon Virginia to Google Taiwan in parallel. Total migration time was around two minutes, but note that most of this time happens in the background without affecting the application. Figure 3.9(b) shows that performance impact was small with a downtime of around 5 seconds. (Downtime could be reduced further by synchronizing the migration finishing time—a project we plan for future work.) The Supercloud maintained the same IP addresses and network topology, without triggering any unnecessary data replication or key shuffling.

ZooKeeper Migration

Experiment setup: To evaluate different follow-the-sun approaches for ZooKeeper, we set up a Supercloud using Amazon EC2 `m3.xlarge` instances (4 vCPUs, 15GB memory) in the Virginia and Tokyo regions. We deployed a ZooKeeper ensemble in three second-layer VMs with 1 vCPU and 1GB memory. The leader was initially in Virginia, with one follower in Virginia and another in Tokyo. We started one client in each region generating a constant workload

with read/write ratio 9:1. Each read operation obtained a 1KB ZooKeeper znode; each write operation overwrote a 1KB znode.

We compared three approaches.

1. A "2-step reconfiguration" moved a majority of the servers from one region to another: We first added a new node in Tokyo, then removed the original leader in Virginia.

   Unfortunately, the 2-step reconfiguration does not guarantee that a node in Tokyo will be elected as the leader. Instead, it was more likely that one of the Virginia nodes would become the leader, which was not desirable since the majority of the ensemble is in Tokyo.

2. A "3-step reconfiguration" ensured that the leader ended up in Tokyo while maintaining the same level of fault tolerance: We added two nodes in Tokyo first, then removed both nodes from Virginia. After the new leader was elected in Tokyo, we added a new node in Virginia and removed one of the nodes from Tokyo.

3. Using the Supercloud we transparently migrated a second-layer VM running the leader from Virginia to Tokyo. Neither ZooKeeper nor clients required any modification.

Experimental results: Figure 3.10 shows the stacked throughput (in operations per second) of clients in both Virginia and Tokyo regions before and after leader migration using the three migration approaches discussed above. After the 2-step reconfiguration shown in Figure 3.10(a), the leader role was switched to a Virginia node while the two followers were in Tokyo, which was inefficient and, as a result, the throughput dropped significantly for both clients. Using

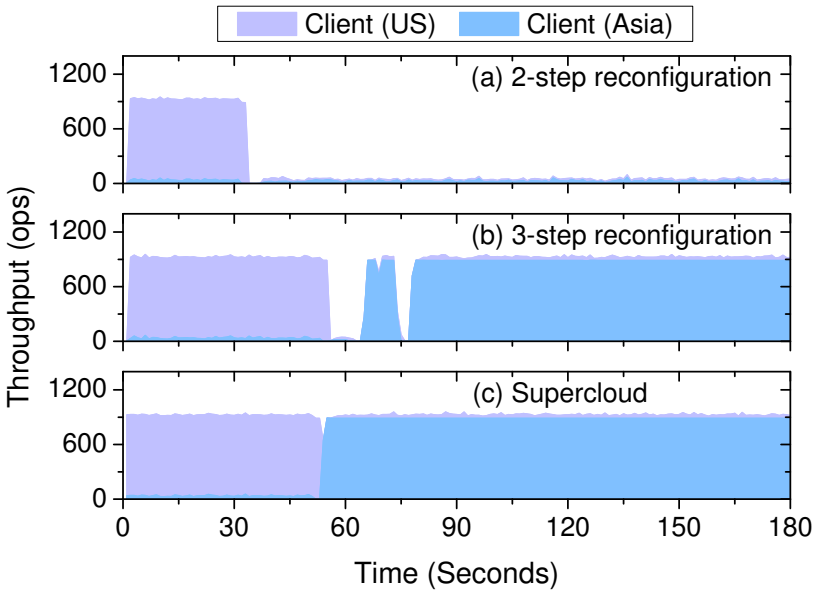


Figure 3.10: Comparison of different migration mechanisms for moving the ZooKeeper leader.

the 3-step reconfiguration in Figure 3.10(b), the leader was successfully moved to Tokyo so good throughput for the Tokyo clients were achieved. Unfortunately, the 3-step reconfiguration took 20 seconds during which performance was inconsistent and low. Finally, Figure 3.10(c) shows the performance of the Supercloud: the drop in performance was less than a second and transparent to the ZooKeeper application and its clients.

3.4.3 Dynamic Resource Scheduling

To examine the efficacy of SDRS, we evaluate it in a private cloud using the TPC-W benchmark. The first-layer VMs have 8GB memory and 8 CPU cores, but we only leave 2.5GB memory and 2 CPU cores for the second-layer Domain-0 and OpenStack Domain-U, and always restrict user VMs to use one other CPU core and 1GB memory. This mimics an environment where first-layer VMs have 3

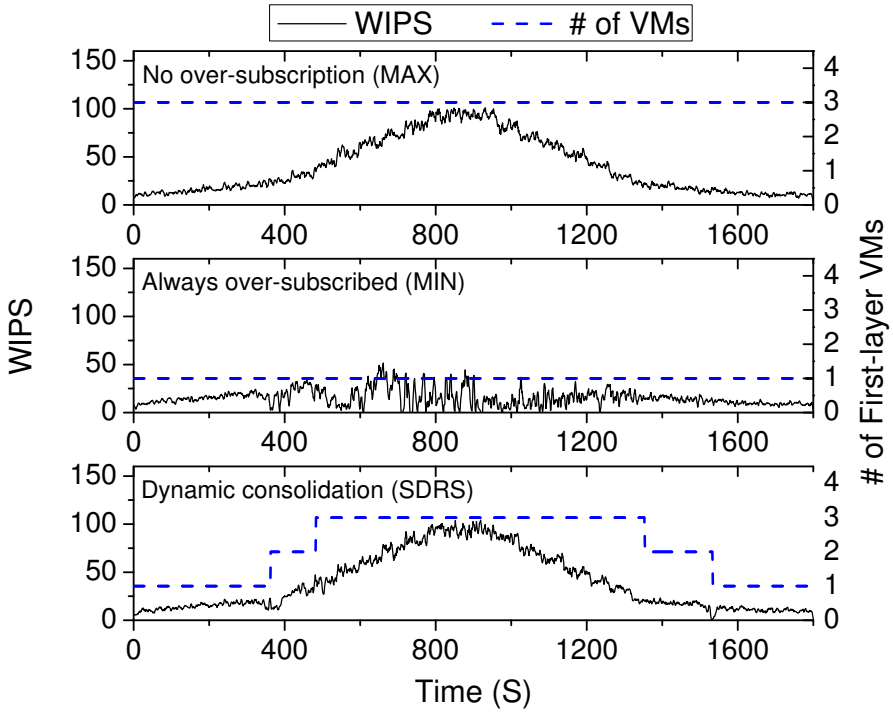


Figure 3.11: Evaluation of SDRS. Dashed lines show the number of first-layer VMs being used.

CPU cores and 3.5GB memory in total. Each second-layer user VM has 1GB memory and 1 CPU core.

In order to decide whether VM consolidation should be performed, the SDRS scheduler needs to estimate future workload of applications. Previous studies have demonstrated that workloads for enterprise applications typically show a periodicity which is a multiple of hours, days, or weeks [63]. It is a common practice to schedule a VM consolidation plan based on the repeating pattern of application workloads. Workload prediction and modeling techniques [111, 127, 107, 124] can also be applied to SDRS. In this experiment, we assume that the workload has a predictable recurring pattern. Three TPC-W clients follow a synthetic workload and feed requests to three TPC-W servers running in different second-layer VMs respectively. This workload is carefully

chosen so that neither the client nor the network can be overloaded.

We compare three scenarios: *no over-subscription (MAX)*, when three second-layer VMs running the TPC-W servers are in separated first-layer VMs, so that resources are always guaranteed; *always over-subscribed (MIN)*, when all three second-layer VMs are packed into one first-layer VM using memory ballooning; *dynamic consolidation (SDRS)*, when SDRS dynamically migrates VMs. Since the workload pattern is known in advance, we use a pre-defined scheduling plan. Memory size of the VMs is adjusted on the fly so that VMs can use as much memory as possible.

Figure 3.11 shows the 5-second moving average of aggregate WIPS (web interaction per second) of all three TPC-W servers. MAX wastes many resources, while MIN significantly hurts the performance. In contrast, SDRS on average uses only 2.13 first-level VMs throughout the experiment, by paying only 1.5% performance degradation. That translates to 29% lower cost than MAX during a half hour experiment.

3.4.4 Storage Evaluation

Experiment setup: To evaluate storage performance under migration, we deployed a Supercloud setup with two `m3.xlarge` instances in Amazon Virginia and Northern California regions. The ping latency between two VMs in these regions was 75ms. We started a user VM with 1 virtual CPU core and 512MB memory and ran the `DBENCH` [6] benchmark in it, which simulated four clients generating 500,000 operations each on the filesystem based on the standard `NetBench` benchmark. We configured the DBENCH clients so that they ran at the fastest possible speed. Without migration, a VM using local storage finished

the benchmark in two minutes.

In the following experiments, immediately after starting the benchmark, we triggered a VM migration from Virginia to Northern California. The full migration took 40 to 50 seconds (most of which was in the background while the VM kept running). During most of the full migration, in particular during the pre-copy phase, the benchmark kept running at the old location. In fact, before the VM was actually moved to Northern California, one third of the workload had finished.

We compared the performance of the benchmark with four different underlying storage systems:

- NFS: A traditional NFS server deployed in Amazon Virginia.
- Sync-on-Write: A strongly consistent geo-replicated store that synchronously propagates each write.
- On-demand: Supercloud storage with proactive propagation turned off.
- Supercloud: Supercloud storage with proactive propagation in the background.

Except for NFS, all schemes were implemented in the Supercloud's propagation manager for fair comparison.

Experiment result: Figure 3.12 shows the results: The average throughput in each second of the DBENCH benchmark. The vertically dashed lines indicate when migration was finished. Figure 3.12a, NFS, shows that throughput dropped significantly after the clients were migrated. Low performance resulted from remote disk accesses that incurred high latency. Figure 3.12b,

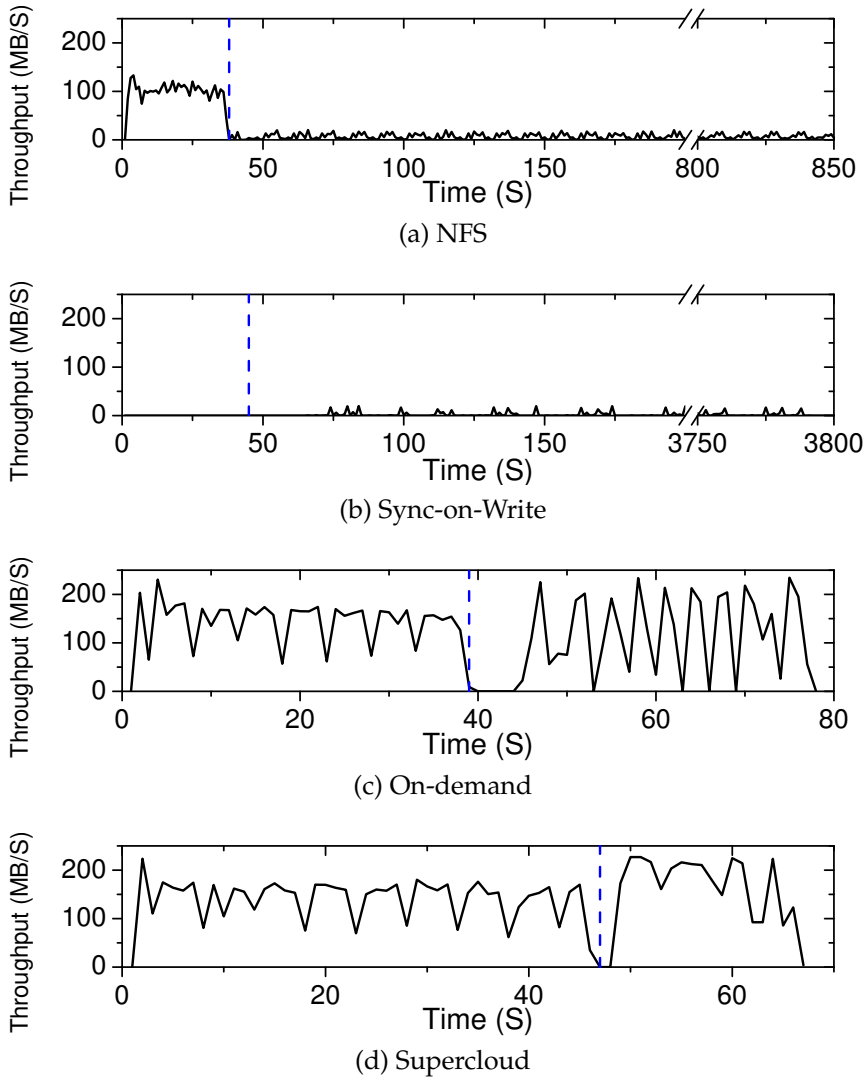
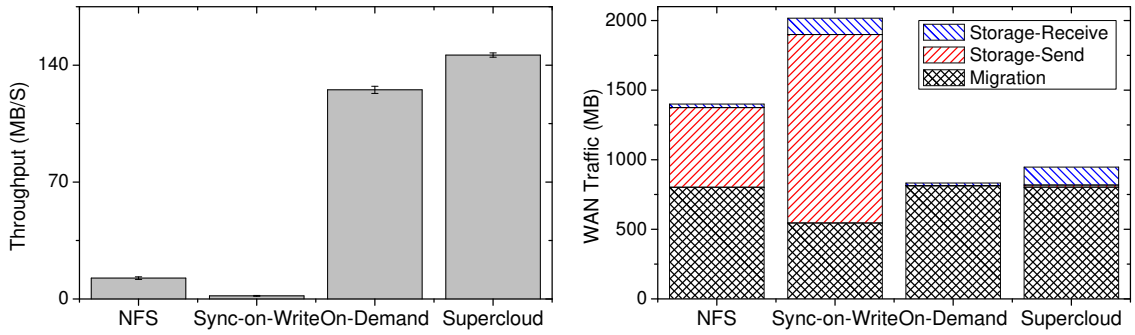


Figure 3.12: Average throughput per second of the DBENCH benchmark in the migrated VM. For clarity, the x-axes are on different scales.

sync-on-write, shows that a strongly consistent geo-replicated storage incurs high performance overhead both before and after migration because each write needs to be propagated through the wide area network. Figure 3.12c, Supercloud on-demand, eventually achieved good average throughput after migration since all writes could be committed locally. However, read throughput was low right after migration since no blocks had been copied ahead of time. Finally, Figure 3.12d, Supercloud proactive propagation, shows that most read



(a) DBENCH throughput (b) WAN traffic on the migration destination

Figure 3.13: Average throughput and total WAN traffic of the DBENCH benchmark in the migrated VM.

and write requests could be served locally. Consequently, the corresponding benchmark took the least time.

Figure 3.13a shows the average throughput for each case. We repeated the same experiment three times and report the average number with standard deviation. The Supercloud proactive propagation achieved the highest throughput.

Figure 3.13b shows the total WAN traffic measured at the migration destination (Northern California). The migration caused 850MB of WAN traffic for NFS, On-demand, and Supercloud, but only 570MB of traffic for Sync-on-Write. Unsurprisingly, the migration traffic depended on the rate at which the VM dirties memory pages. A benchmark running at a higher speed caused more dirty memory pages to be copied during migration. Sync-on-Write storage results in low throughput in the benchmark, thus lowering the page dirty rate of the whole VM. Both NFS and Sync-on-Write incur many outgoing traffic when accessing storage because each disk write needed to go through the WAN. Sync-on-Write also had higher incoming traffic, which was generated when the benchmark was running in Virginia. On-demand achieved the lowest

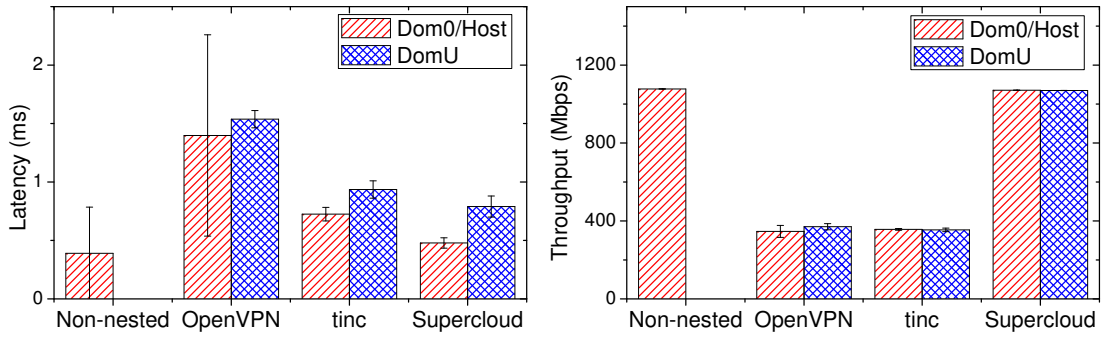
WAN traffic since it fetched data remotely only when necessary, and no update propagation was needed. The Supercloud incurred more incoming traffic than On-demand because it pushed some data that was not needed, but it achieved higher throughput and predictable performance.

3.4.5 Network Evaluation

To evaluate the network performance of the Supercloud, we deployed the Supercloud in two `m3.xlarge` instances in the Amazon Virginia region and measured UDP latency (using UDP ping) and TCP throughput (using `netperf`) between second-layer Domain-0 VMs and Domain-U VMs respectively. For comparison, we ran the same benchmark using the following settings:

- Non-nested: a setup where we ran the benchmark directly in the first-layer VMs for baseline purposes.
- OpenVPN: a VPN solution using a centralized controller, also running in Amazon Virginia.
- tinc: a P2P VPN solution, which implements full-mesh routing.
- Supercloud: the Supercloud implementation based on Open vSwitch.

For the latency test, we ran 50 UDP pings (1 ping per second). Figure 3.14a shows the average latency and standard deviation across all runs. Although our Open vSwitch-based virtual network slightly increases the UDP latency, the overhead was much smaller than either OpenVPN or tinc. The latency for Dom0 was smaller than DomU because network packets to DomU go through Dom0 first. OpenVPN had the highest latency because all packets travel through the centralized controller.



(a) UDP latency (b) TCP throughput

Figure 3.14: Network performance evaluations.

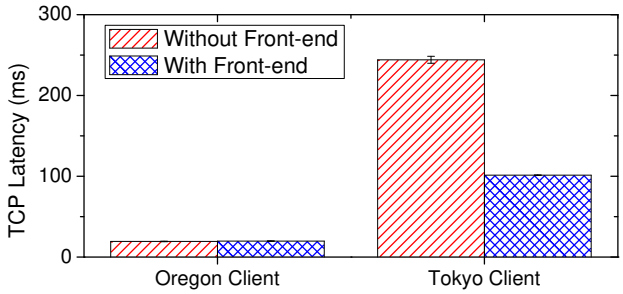
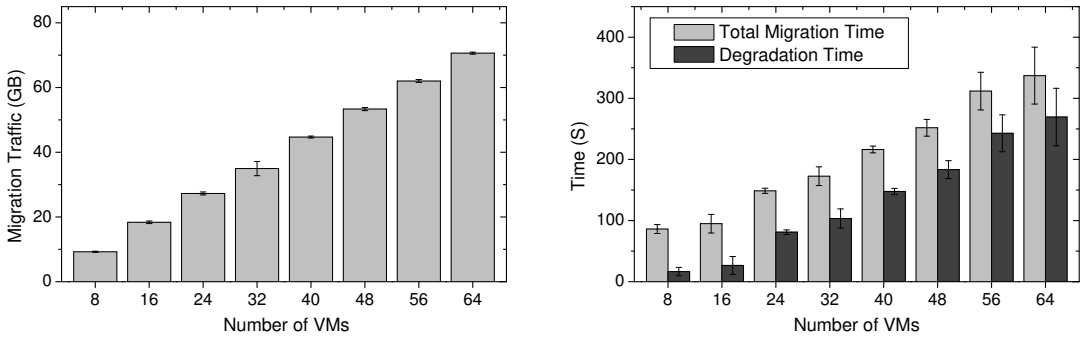


Figure 3.15: Impact of using public IP front-ends.

To measure TCP throughput, we enabled jumbo frames in all setups and repeated a `netperf` TCP stream benchmark with default options 10 times for each setup. As shown in Figure 3.14b, the throughput of non-nested instances, Dom0, and DomU in the Supercloud all achieved 1Gbps with small variance. In contrast, tinc and OpenVPN could only achieve 300Mbps. Because both tinc and OpenVPN use a tap device connected to a user-level process, the extra memory copy and kernel-user mode context switching incurred significant overhead.

To evaluate the impact of using public IP front-ends, we deployed a second-layer VM as a server running in the Amazon Oregon region. We had two clients running in Google Compute Engine Oregon and Tokyo regions respectively and used the `hping` tool to measure TCP latency from clients to the server with and



(a) Migration traffic

(b) Migration time and period of performance degradation

Figure 3.16: Migration time and traffic with different number of VMs.

without public IP front-ends. Each test was repeated 15 times. Without public IP front-ends, the clients used the public IP address of the first-layer VM directly to communicate with the server. With public IP front-ends, each client communicated with a front-end running in a nearby Amazon data center. Figure 3.15 shows the average TCP latency from different clients to the server in both cases. For the Oregon client, the overhead of adding a front-end was negligible. Interestingly, the Tokyo client measured lower latency by going through the public IP front-end in Tokyo. This was because the front-end was running in an Amazon data center, and the latency from Amazon Tokyo to Amazon Oregon was significantly less than the latency from Google Tokyo to Amazon Oregon.

3.4.6 Scalability

In this section, we evaluate scalability of Supercloud live migration by migrating a Cassandra cluster with varying numbers of nodes. We set up a Supercloud in the Amazon EC2 Oregon and Tokyo regions. In each region we deployed 16 `c4.xlarge` nodes (4 vCPUs, 7.5GB memory) for running second-layer VMs. We chose 16 as the cluster size in each region because for an ordinary user Ama-

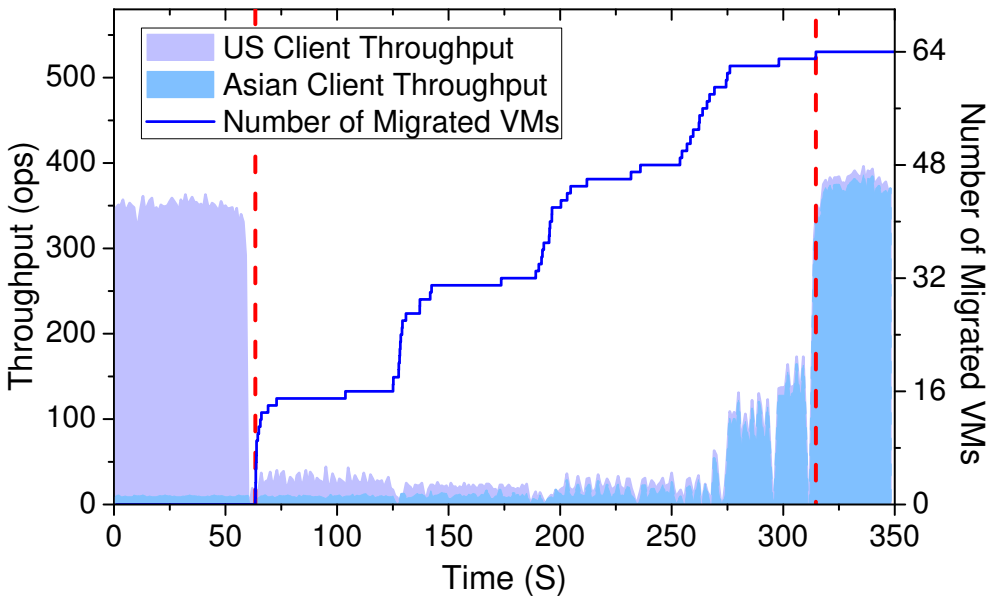


Figure 3.17: Workload trace of migrating a 64-VM Cassandra cluster.

zon imposes a default limit of 20 on the number of EC2 instances in a single region, and we also had to run other instances for storage and workload generators.

We started 8 to 64 second-layer VMs for running Cassandra. Each second-layer VM had 1GB memory and 1 vCPU. The key space was evenly distributed across the whole cluster, and the replication factor was set to 2. The workload generator was the same as what was used in Section 3.4.2: each region had a client generating read and write operations on random keys with a ratio of 4:1. The Cassandra cluster was initially in the Oregon region.

Figure 3.16 shows the network traffic and time taken to migrate the entire Cassandra cluster from the Oregon region to the Tokyo region while running the workload generator. Network traffic increases approximately linearly with the number of second-layer VMs because we need to copy at least 1GB memory for each second-layer VM. It is possible to reduce total migration time and traffic

by deduplicating the copied memory pages, which is left for future work.

"Degradation Time" shows the time during which the total throughput of all clients dropped by more than 50%. We can see that the degradation time increased with the number of VMs. This is because the workload generator we used issued synchronous requests one by one to random Cassandra nodes. Any time the cluster was split in two different locations, throughput of the workload was affected significantly. Figure 3.17 shows the stacked workload trace of the US and Asian clients when the cluster size was 64. The vertical dash lines indicates the time when the first and last VM was migrated. As we can see the total throughput dropped until the whole cluster was moved. As part of future work, we want to develop an "ensemble live migration" mechanism where the foreground stop-and-copy phase of the migration is synchronized across virtual machines. Doing so should significantly reduce the time during which performance is degraded.

3.5 Summary

A Library Cloud is a cloud that can seamlessly span multiple cloud providers and support user-level resource management. The Supercloud is an instance of a Library Cloud and presents a complete cloud software stack under the user's full control that can seamlessly span multiple availability zones and cloud providers, including private clouds. It features live migration, as well as shared storage, virtual networking, and automated scheduling of workloads, placing and migrating VM resources as needed. Spanning availability zones and cloud providers, the Supercloud provides maximal flexibility for placement. Using our automated schedulers, we demonstrate continuous low latency for diurnal

workloads that it is important for global cloud services to be able to “follow the sun”.

CHAPTER 4

RELATED WORK

In this chapter, we discuss work related to our contributions, the techniques they build on, and alternate or complementary approaches.

4.1 X-Containers

Table 4.1 summarizes properties of the X-Container architecture and alternative approaches. Below we discuss why those other approaches cannot satisfy all of our design goals.

OS-level virtualization: OS-level virtualization [110] provides a lightweight mechanism to support containers with the illusion of a dedicated OS. Docker [7], LXC [12], OpenVZ [18], and Solaris Zones [101] are different implementations of OS-level virtualization. Generally these solutions provide poor kernel customization support, and application isolation is a concern due to the sharing of a large OS kernel.

	X-Container	Linux Containers	Virtual Machine	Unikernel, EbbRT, OSv	LibraryOS (Exokernel)	Graphene	UML	Dune	SCONE
Small TCB and Attack Surface	✓	✗	✓	✓	✓	✗	✗	✗	✓
Optimized Kernel Access	✓	✗	✗	✓	✓	✓	✗	P[4]	✓
Kernel Customization	✓	✗	✓	✓	✓	✓	✓	✓	✓
Full Linux ABI Support	✓	✓	✓	✗	✗	P[1]	✓	✗	✗
Multi-Process Support	✓	✓	✓	✗	✗	✓	✓	✗	✗
Portability	✓	✓	P[2]	P[3]	✓	✓	✓	✗	✗
Scalability	✓	✓	✗	✓	✓	✗	✗	✓	✗

[1] Only supports about 1/3 of Linux system calls.
[2] Requires hardware virtualization support except Xen-Blanket PV instances.
[3] Need to compile to different images for different clouds.
[4] Some system calls are served by the host kernel by using `VMEXIT` instructions.

Table 4.1: Comparing X-Container with alternative approaches (✓-Fully Supported; P-Partially Supported; ✗-Not Supported)

Library OS: The concept of a Library OS [57, 100, 29, 50, 87] is to keep the kernel small and link an application with functions that are traditionally performed in the kernel. Most Library OSs [57, 27, 69, 100, 103] focus on single-process applications, which is not sufficient for supporting multi-process container environments, nor can they support Linux applications in general. Graphene [115] is a Library OS that supports multiple Linux processes, but it does not provide full Linux application binary interface (ABI) support. (For example, only one third of Linux system calls are supported.) Moreover, multiple processes use IPC calls to access a shared POSIX implementation, which limits its performance and scalability. Finally, the underlying host kernel of Graphene is a full-fledged Linux kernel, which does not reduce the TCB and attack surface.

Virtual Machines: Virtual machines [52, 113, 33, 82] support packaging applications and the OS into a single unit. Xen-Blanket [122] enables portable paravirtualization of nested VMs in public clouds. However, running applications in full-fledged VMs is not scalable and resource-efficient because they also run many unrelated processes and services. In addition, the OSs in VMs introduce unnecessary isolation overhead when processes are trusting each other.

Lightweight Virtual Machines: Unikernel [89] and EbbRT [105] can compile application source code directly into a lightweight virtual machine running in the cloud. Similarly, OS^v [19] is a language runtime that runs an application in a single address space with a lightweight OS in a VM. Unlike Unikernel, EbbRT, and OS^v, which only support single process and require re-writing or re-compiling the application, X-Containers support multi-process and binary level compatibility, and can be immediately deployed to all major cloud platforms. In addition, X-Container supports all debugging and profiling features that are

available in Linux.

User Mode Linux (UML): User mode Linux (UML) [54] allows running a Linux kernel in user space and supports multiple processes. Similar to Graphene, UML runs on top of a full-fledged Linux kernel, not reducing the TCB or attack surface. Moreover, the address space of the UML kernel and its user processes are isolated. Interrupts are implemented as signals and system calls are implemented with `ptrace`, both at significant overhead. Usermode Kernel [61] is an idea similar to X-Containers that runs parts of the 32-bit Linux kernel in userspace in VM environments. However, some parts of the Usermode Kernel still run in a higher privilege level than user mode processes, and it is not integrated with application container environments.

Dune: Dune [34, 35] is a system that runs a process with a LibOS using hardware virtualization. Dune processes are tightly coupled with the underlying Linux kernel performing system calls with `VMEXIT` instructions, while X-Containers run on an exokernel. Dune is based on a process isolation model, while X-Containers isolate groups of processes efficiently sharing the same LibOS. Although Dune provides a modified version of libc, it is not sufficient to achieve the same level of compatibility as X-Containers, which can run applications, containers, and even Linux kernel modules unmodified. Finally, Dune cannot run within a VM in the public cloud because it requires nested hardware virtualization support, which public clouds do not expose.

SCONE: SCONE [31] implements secure containers using Intel SGX. However, all SCONE containers still share the same kernel, a single point of vulnerability. Moreover, due to hardware limitations, SCONE cannot support running

multiple processes within a container. Finally, SCONE requires relinking the application to a special Library OS.

4.2 Library Cloud

4.2.1 Multi-Cloud Deployment

Multi-cloud deployment is attractive to users because of its high availability and cost-effectiveness. SafeStore [83], DepSky [40], HAIL [44], RACS [26], SPANStore [126], Hybris [55] and SCFS [41] have demonstrated benefits of multi-cloud storage, but they do not support computational resources. Docker [7] deploys applications encapsulated in Linux containers (LXC) to multiple clouds. Although light-weight, LXC is not as flexible as a VM because all containers must share the same kernel, and it has poor migration support. Ravello [20] leverages a nested hypervisor to provide an encapsulated environment for debugging and development distributed applications, but it does not addresses the challenges of providing storage and network support for wide-area application migration. Platforms like fos [120], Rightscale [21], AppScale [4], TCloud [118], IBM Altocumulus [91], and Conductor [121] enable multi-cloud application deployment, but none of them provides the generality and flexibility of a multi-cloud IaaS.

4.2.2 Wide-area VM Migration

VM live migration [51] has been widely used for resource consolidation and workload burst handling [72, 95, 46]. A traditional VM live migration only

involves memory transfer and assumes the disk image is shared. The pre-copy strategy [51] is the most widely used memory transfer technology. Post-copy [73] has been proposed to eliminate duplicated transmission in pre-copy. However, VM live migration is not exposed to end users of public clouds.

Migrating a VM in the wide-area network faces long latency in accessing a shared disk image. To address this problem, Bradford et al. [45] propose to iteratively copy the image in parallel with the memory. CloudNet [123] presents optimizations that minimize the cost of storage transfer and memory. Hirofuchi et al. [74] combine on-demand fetching and a background copy after the memory is migrated. Mashtizadeh et al. [90] describe various solutions used for live storage migration in VMware ESX. Zheng et al [129] optimize storage migration by leveraging temporal and spatial locality. Nicolae et al. [96] propose a hybrid local storage transfer scheme for live migrating VMs with I/O intensive workloads—it combines pre-copy, post-copy, and prioritized prefetching based on access frequency. CloudSpider [42] combines VM image replication and scheduling—it replicates the VM image asynchronously in the background, but the image needs to be synced before migration is finished. Seagull [67] facilitates cloud bursting by determining which applications can be transferred into the cloud most economically—it uses opportunistic pre-copy to transfer an incremental snapshot of a VM's disk state. Our proactive approach attempts to only transfer data that is critical to performance.

VMFlockMS [28] exploits similarity among VM images. VMFlockMS focuses on offline VM migration, requiring to shut down the VM first. FVD [114] is a new VM image format that supports copy-on-write, copy-on-read, and adaptive prefetching that can be used for optimizing the performance of migrating a

VM with the disk image. However, FVD can prefetch data only after the VM resumes on the destination. In contrast, our Supercloud storage proactively propagates data before migration is triggered.

4.2.3 Dynamic Resource Scaling

Resource scaling has been widely studied. Fine-grained resource scaling solutions adjust CPU, memory, and I/O resource allocation based on control theory [130, 79, 98], workload prediction [107, 48], or workload modeling [112, 56]. Coarse-grained capacity scaling schemes dynamically adjust the number of nodes in a distributed system [88]. VM cloning [84] and live migration are also widely used for resource consolidation or workload burst handling [72, 95, 46]. Techniques such as fine-grained CPU capping or VM live migration are typically not available to cloud end-users. In addition, due to the complexity of different applications, it is impractical for cloud providers to provide services that can fit all requirements of any application. In contrast, users have much more knowledge about the application and its requirements. In the Supercloud users have control over their applications and have the flexibility and control in choosing resource scaling solutions. All these techniques are compatible with the Supercloud and indeed easier to support than in traditional clouds, which would require the cloud provider to expose such functionality.

4.2.4 Nested Virtualization

Nested virtualization has been studied and theoretically analyzed in 1970s [64, 65, 99]. Belpaire et al. [36] created a formal model for recursive virtual machines which needs a centralized supervisor. Lauer et al. [86] removed this require-

ment and proposed to leverage nested virtual memories to create nested virtualization environment. Belpair et. al. [37] presented a formal model of hardware/software architecture which can be applied to recursive virtual machine systems. The IBM z/VM hypervisor [97] is the first practical implementation of nested virtualization, which relies on multi-level architectural support. These solutions typically require hardware mechanisms and corresponding software support which bear little resemblance to x86 architecture and operating systems.

A traditional x86 architecture has only a single level of architectural support for virtualization. Ford et al. [58] proposed to enable nested virtualization on x86 platforms by modifying the software stack at all levels based on a microkernel. Their goal is to enhance OS modularity flexibility and extensibility rather than virtualizing legacy OS'es. As new hardware extensions are being added into x86 platforms such as Intel-VT [116] and AMD-V [3], people start searching for nested virtualization solutions that can utilize hardware primitives at different levels. The Turtles project [38] extended KVM with nested virtualization support on Intel processors. It also improves the I/O performance by enabling multi-level device assignment. Alexander et al. [66] implemented nested virtualization based on AMD processors in KVM. Recently nested virtualization supports have been proposed for Xen hypervisor [70]. CloudVisor [128] protects the privacy and integrity of customers' virtual machines on commodity virtualized infrastructures by introducing a tiny security monitor underneath the commodity VMM and exposing nested virtualization support for guest VMs. Although these nested virtualization solutions share many motivations with Xen-Blanket, their major focus is how to support multi-level full virtualization with current hardware primitives. In contrast, Xen-Blanket seeks for a practical solution for nested para-virtualization, which has technical challenges fundamentally dif-

ferent with that of exposing hardware support to multiple levels.

Blue Pill [60] is a root-kit emulating VMX in order to remain functional and avoid detection when a hypervisor is installed in the system. it is loaded during boot time by infecting the disk master boot record (MBR). Its nested virtualization support is minimal since it only needs to remain undetectable. In contrast, a comprehensive nested virtualization platform must efficiently multiplex the hardware across multiple levels of virtualization, and manage all CPU, MMU, and I/O resources. Berghmans [39] proposes another nested x86 virtualization platform where a software-only hypervisor is running on a hardware-assisted hypervisor. Different with this approach, Xen-Blanket does not assume any hardware virtualization support.

CHAPTER 5

FUTURE DIRECTIONS

By separating protection and management, cloud infrastructures can improve security, flexibility, and efficiency. This dissertation applies this approach to two important platforms in cloud infrastructures: the containers platform, and the IaaS platform. In this chapter, we discuss some of the possible directions this technology may take in the future, and the problems relevant to those directions.

5.1 X-Container-Based Supercloud

Superclouds leverage nested virtualization solutions such as Xen-Blanket to provide a homogeneous computation environment across different cloud providers. Nested virtualization incurs non-negligible overhead, especially in x86-64 environments, as discussed in Chapter 2. The X-Container platform can efficiently run in virtualized environments without introducing significant overhead. Further, X-Containers run on the X-Kernel, which is essentially a VM hypervisor that supports all existing hypervisor-level operations, such as live migration, memory page sharing, and consolidation. These are compelling reasons to replace the nested virtualization layer in the Supercloud with the X-Containers platform and evaluate its performance and benefit.

5.2 X-Containers for Serverless Computing

Serverless computing is a new cloud service model in which cloud providers execute user programs in response to pre-defined events. Users are charged

based on actual resource consumption instead of pre-allocated capacity. There are many advantages to the serverless computing model. For example, it is more cost-effective than traditional cloud service models since it does not require preserving a fixed amount of resources. Developers do not need to spend time setting up the environment and tuning system parameters. It also simplifies programming because typically user-defined functions are single-threaded.

Most existing serverless computing platforms use containers as the underlying mechanism for packaging and deploying user functions. However, due to the concern of container security isolation, they often run containers in VMs, which sacrifices performance and resource efficiency. X-Containers can potentially provide a solution for serverless computing that not only improves the security isolation of different users, but also improves performance and resource efficiency by supporting kernel customization. However, there are still many challenges that need to be addressed. For example, spawning a new X-Container can take two to three seconds, which is too slow for serverless computing scenarios where containers are initialized and destroyed in milliseconds. Further, X-Containers require a more efficient mechanism to share physical resources such as memory. Although X-Containers can use memory ballooning to adjust memory allocation, it is not as efficient as sharing memory among different processes running on the same kernel.

5.3 Linux kernel optimized for X-Containers

The X-Container architecture enables great flexibility for customizing the OS kernel for applications. For example, users can now install kernel modules as needed, and they have the freedom to tune any kernel parameter. Further opti-

mization is possible given the fact that there is no isolation between the kernel and the user program.

The existing Linux kernel is designed and implemented based on the assumption that user mode programs are untrusted. The mechanism used to send notifications from the kernel to user programs involves many security checkpoints. For example, signals, which represent one important method for implementing kernel call-back functions, have significant overheads since using them requires copying the stack between the kernel mode and the user mode for security concerns. Other mechanisms require ad-hoc system call interfaces such as `ioctl` or `inotify`, and may introduce a non-negligible scheduling delay. All these overheads can be eliminated in X-Containers by opening new interfaces for kernel callback functions. This enables many interesting design choices. For example, we can implement kernel modules with high-level languages such as Python and OCaml. We can call many user mode libraries even in the device driver. For applications with low latency requirements, we can allow a direct callback from the device driver with minimum overhead.

5.4 Efficient resource sharing and communication among X-Containers

X-Containers are designed to enhance isolation among containers running on the same physical host or VM. In a scenario where multiple containers are cooperating with each other, efficient resource sharing becomes critical. One possible solution is found in Docker, which supports sharing file systems with the host and other containers. We can also use a pipe to connect applications running in different containers.

It is more challenging to implement efficient resource sharing and communication in X-Containers than in Docker containers, because X-Containers only share an exokernel, which implements very basic resource multiplexing and security isolation. In contrast, Docker containers share a full-fledged monolithic Linux kernel which provides many mechanisms for resource sharing and interprocess communications. In many cases, we can treat multiple X-Containers as a distributed system and implement resource sharing with network functions. For example, a shared file system can be implemented with NFS (Network File System). However, this solution clearly introduces unnecessary overheads on the network stack.

A better way to implement efficient resource sharing and communication is by leveraging the low-level inter-container communication mechanism supported by the exokernel. In the X-Container architecture, we can do this by using hypercalls and memory page sharing. For example, we can provide a shared file system which uses hypercalls to synchronize the page cache in different X-Containers. The page cache can be even shared to further improve memory utilization. As another example, we can provide a special pipe to connect processes running in different X-Containers. The interface of the pipe is the same as existing Linux pipes, but the underlying implementation can be based on hypercalls and memory sharing. It is interesting to investigate how mechanisms implemented with hypervisor-level primitives compare to mechanisms implemented with network-level communications.

5.5 Online Vertical Scaling in the Library Cloud

One of the key motivations for users to adopt the cloud computing paradigm is the elasticity enabled by horizontal scaling, that is, dynamically adding or removing virtual machines (VMs) used by the application in response to workload fluctuations. Unfortunately, many applications do not support horizontal scaling. In order to take advantage of dynamic cluster size, an application needs to be programmed carefully to deal with load balancing, state replication, distributed transactions, and fault tolerance. Non-distributed applications (e.g., a single MySQL database or an SVN repository) and distributed applications with static memberships (e.g., ZooKeeper) cannot easily leverage horizontal scaling. Even for applications that support horizontal scaling by, say, changing the number of shards, the overhead of re-sharding can be substantial. It would be ideal if cloud providers supported another type of elasticity: vertical scaling, that is, dynamically changing the size of a VM in terms of the number of CPU cores, memory size, and I/O throughput.

Existing public cloud providers only support offline vertical scaling, which requires that users turn off a VM before changing its size. Offline vertical scaling has three major drawbacks: first, it is currently triggered manually; second, it incurs a long service downtime; and third, it requires applications to re-initialize the internal state and configurations. Because online vertical scaling is not supported in IaaS clouds, many VMs are kept idle in clouds, not because they are running all the time, but because they are either supposed to be online 24 hours a day or it is too hard for the user to re-initialize the state after a shutdown. In this case, users are overpaying while resources are wasted.

There are several reasons why existing cloud providers do not support online vertical scaling. First, it complicates resource management. Increasing the size of a VM on a physical server might prevent it from accepting new VMs, which makes it hard to optimize resource utilization. Second, it introduces additional complexity. Suppose the physical server on top of which the VM is running does not have sufficient resources to satisfy a vertical scaling request. Now the VM has to be live migrated, which incurs extra CPU, memory, and I/O costs. Cloud providers do not have incentive to overcome these challenges, because users are paying for their resource reservations anyway, even when they are under-utilized.

We believe that online vertical scaling is a missing piece of elasticity in IaaS clouds that can be implemented without introducing any extra complication or overhead to cloud providers. Leveraging online vertical scaling, users can immediately get benefits of vertical scaling without waiting for any cloud provider to give specific support. We also believe that vertical scaling should be automatic, so that costs can be saved without the involvement of cloud users while keeping the performance impact to applications minimal.

Online vertical scaling can be implemented in the Library Cloud, since users of the Library Cloud have full control of the cloud stack. The basic approach is to migrate applications live across VMs with different sizes. There are many challenges that need to be addressed:

How to minimize performance overhead of nested virtualization? Existing nested virtualization technologies focus on running a full-fledged hypervisor inside a VM, with all security isolation mechanisms. Our observation is that depending on the usage model, a nested hypervisor does not need to enable

all isolation mechanisms. In the vertical scaling scenario, nested VMs and the nested hypervisor are trusting each other if two nested VMs must be isolated, we can simply run them in different underlying VMs. So in this case a nested hypervisor should focus on providing device virtualization and enabling migration. We can provide a lightweight nested virtualization platform that is highly optimized for this purpose with minimal performance overhead.

How to support migration with local storage? Public clouds allow VMs to allocate high-speed temporary storage. (Typically provisioned using local SSDs.) Migrating an application across different VMs implies that the local storage has to be moved as well. If we copy all data in the local storage when performing migrations, it can cause long migration time and significant performance overhead. On the other hand, serving the temporary storage remotely through the network can eliminate the need for migration, but this sacrifices performance benefits and adds network costs. We propose to provide a virtual SSD that is implemented as a distributed disk with data in both local and remote disks. Multiple levels of cache can be leveraged to match the performance of a real SSD. The local disk can transfer data to the remote disk in order to minimize data copy when performing migrations. Another direction we can explore is to expose a weak consistency model in the temporary storage. Studies [108] have demonstrated that many applications can tolerate a weak consistency model for temporary data. By relaxing the consistency model, data copy can be further reduced when migration is performed.

How to make online vertical scaling automatic? The challenge here is how to provide an interface that allows users to easily customize the automatic scaling policies for different types of applications. We can leverage the state machine

model, with which users can specify multiple predefined instance types, along with a set of transition rules to determine the next state of an instance. The transition rules can be based on a performance goal or a resource pressure threshold. Another interesting direction is to investigate monitoring and handling resource pressure propagation and bottleneck shifting.

CHAPTER 6

CONCLUSION

At the time of writing this dissertation, cloud infrastructures typically sacrifice efficiency and flexibility in order to guarantee security. Containers are run in virtual machines, losing the benefit of resource efficiency, and kernel customization for containers is not supported. IaaS platforms do not expose useful administrative APIs, limiting the flexibility of migrating computation across different locations and optimizing resource utilization.

We explored the approach of separating protection and management, a fundamental principle behind the exokernel architecture that was proposed to improve traditional operating systems. By separating protection and management, the protection layer can focus only on security isolation and resource multiplexing, making security guarantee easier to maintain and verify. Resource management components are dedicated to each user or application for customization and optimization, greatly improving flexibility and efficiency.

We applied the approach of separating protection and management to containers and IaaS platforms, and presented X-Containers and Library Cloud. X-Containers provide strong security isolation and kernel customization support to containers, while still supporting efficient container execution. Library Cloud is a new abstraction that enables more flexible and efficient user-level resource management without breaking security isolation between different users.

Cloud infrastructures are evolving all the time, with new service models emerging and new architectures developing. But the requirement of security, flexibility, and efficiency remains constant. This dissertation demonstrates that

the principle of separating protection and management is effective in practice on improving security, flexibility, and efficiency, and is widely applicable to different layers in the cloud infrastructure.

GLOSSARY

A

API Application Programming Interface

C

cloud The data center hardware and software for cloud computing. See *cloud computing*.

cloud computing Both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services [30].

cloud infrastructure The collection of hardware and software that enables cloud computing.

cloud platform See *cloud infrastructure*.

cloud stack The architecture of a cloud with different layers exposing different abstractions.

containers User-space instances virtualized by the OS kernel that have separated views on the file system, network stack, user IDs and groups, and processes.

D

data center A large group of networked computer servers typically used by organizations for the remote storage, processing, or distribution of large amounts of data.

E

efficiency The capability of improving performance and reducing cost of resources including hardware, energy, man power, money, and time.

elastic scaling The ability of a cloud service provider to provision flexible computing power when and wherever required. The elasticity of these resources can be in terms of processing power, storage, bandwidth, etc.

F

flexibility The support of user customization on cloud services and resource management policies.

H

hypervisor Computer software, firmware or hardware that creates and runs virtual machines [47].

I

Infrastructure as a Service (IaaS) A cloud service model that supports users to deploy and run arbitrary software including operating systems and applications, and provides limited control of networking and storage components [93].

Internet The global system of interconnected computer networks that use the Internet protocol suite (TCP/IP) to link devices worldwide.

isolation Security isolation. See *security*.

IT Information Technology

M

maintenance A process that is typically performed periodically to preserve an asset's operational status and original condition, compensating for normal wear and tear.

multi-tenancy The capability with which a single instance of a software system serves multiple tenants.

N

NIST National Institute of Standards and Technology

O

operating system (OS) System software that manages computer hardware and software resources and provides common services for computer programs.

OS kernel The central component of an operating system that manages all physical resources and provides low-level abstractions for programs.

P

platform The environment in which a piece of software is executed. It includes hardware, the operating system (OS), and other software executing in it.

Platform as a Service (PaaS) A cloud service model that supports users to deploy onto the cloud infrastructure consumer-created or acquired applications, using programming languages, libraries, services, and tools supported by the provider [93].

private cloud The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.

private data center One type of data center that is operated solely for a single organization. See data center.

provisioning (resource provisioning) The process of preparing computation resources for serving users, such as allocating servers, bootstrapping software systems, and connecting network components.

public cloud The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.

S

scalability The capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.

security The protection of user computation and resources against unauthorized access, data leakage, and malicious attacks or damages.

security isolation See *security*.

serverless computing A cloud computing execution model in which clouds provide runtime environment to execute user programs on-demand, and dynamically manage the allocation of physical resources.

Software as a Service (SaaS) A cloud service model that supports users to use the provider's applications running on a cloud infrastructure [93].

Software-defined Networking (SDN) A technology that allows network administrators to initialize, control, change, and manage network behavior dynamically via open interfaces.

T

Trusted Computing Base (TCB) A small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.

V

virtual machine (VM) An emulation of a computer system that provides a substitute for a real machine and functionalities needed to execute entire operating systems.

Virtual Private Network (VPN) A network that extends a private network across a public network, and enables users to send and receive data across shared or public networks as if their computing devices were directly connected to the private network.

virtualization The application of the layering principle through enforced modularity, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized [47].

VM consolidation A process of packing a set of running VMs on the same physical machine to share resources and improve resource utilization.

VM live migration A process of moving a running VM between different physical machines without disconnecting or interrupting the clients and applications.

BIBLIOGRAPHY

[1] Amazon Auto Scaling. http://aws.amazon.com/autoscaling.

[2] Amazon EC2. http://aws.amazon.com/ec2/.

[3] AMD Virtualization. http://www.amd.com/en-us/solutions/servers/virtualization.

[4] AppScale: The Open Source App Engine. http://www.appscale.com.

[5] Cassandra Documentation: Replacing a Dead Node. https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_replace_node_t.html.

[6] DBENCH benchmark. https://dbench.samba.org/.

[7] Docker. https://www.docker.com.

[8] GNU GRand Unified Bootloader (GRUB) documentation. https://www.gnu.org/software/grub/grub-documentation.html.

[9] Google Compute Engine. https://cloud.google.com/compute/.

[10] Google Drive, Dropbox, Box and iCloud Reach the Top 5 Cloud Storage Security Breaches List. https://psg.hitachi-solutions.com/credeon/blog/google-drive-dropbox-box-and-icloud-reach-the-top-5-cloud-storage-security-breaches-list.

[11] Intel patches remote hijacking vulnerability that lurked in chips for 7 years. https://arstechnica.com/information-technology/2017/05/intel-patches-remote-code-execution-bug-that-lurked-in-cpus-for-10-years/.

[12] Linux LXC. https://linuxcontainers.org/.

[13] List of Security Vulnerabilities in the Linux Kernel. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html.

[14] Microsoft Azure Cloud Computing Platform & Services. https://azure.microsoft.com/.

[15] Open vSwitch. http://openvswitch.org.

[16] OpenStack. http://www.openstack.org/.

[17] OpenVPN. https://openvpn.net/.

[18] OpenVZ Containers. https://openvz.org/Main_Page.

[19] OSv operating system. http://osv.io/.

[20] Ravello Systems. http://www.ravellosystems.com/.

[21] Rightscale. http://www.rightscale.com.

[22] Rumprun Unikernel. https://github.com/rumpkernel/rumprun.

[23] tinc VPN. http://www.tinc-vpn.org/.

[24] XenServer. http://www.xenserver.org/.

[25] XenServer-Core. https://github.com/xenserver/buildroot.

[26] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 229–240, New York, NY, USA, 2010. ACM.

[27] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. 1986.

[28] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. VMFlock: Virtual Machine Co-migration for the Cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 159–170, New York, NY, USA, 2011. ACM.

[29] T. E. Anderson. The case for application-specific operating systems. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*, pages 92–94, Apr 1992.

[30] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[31] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 689–703, GA, November 2016. USENIX Association.

[32] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[33] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, New York, NY, USA, 2003. ACM.

[34] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[35] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[36] Gerald Belpaire and Nai-Ting Hsu. Formal properties of recursive Virtual Machine architectures. In *Proceedings of the fifth ACM Symposium on Operating Systems Principles (SOSP'75)*, pages 89–96, New York, NY, USA, 1975. ACM.

[37] Gerald Belpaire and Nai-Ting Hsu. Hardware architecture for recursive

Virtual Machines. In *Proceedings of the ACM Annual Conference (ACM'75)*, pages 14–18, New York, NY, USA, 1975. ACM.

[38] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles project: design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[39] Olivier Berghmans. Nesting virtual machines in virtualization test frameworks. Master's thesis, Department of Mathematics and Computer Science of the Faculty of Sciences, University of Antwerp, 2010.

[40] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-clouds. In *Proceedings of the Sixth European Conference on Computer Systems (EuroSys'11)*, pages 31–46, New York, NY, USA, 2011. ACM.

[41] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A Shared Cloud-backed File System. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, Philadelphia, PA, June 2014. USENIX Association.

[42] Sumit Kumar Bose, Scott Brock, Ronald Skeoch, and Shrisha Rao. CloudSpider: Combining Replication with Scheduling for Optimizing Live Migration of Virtual Machines Across Wide Area Networks. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

[43] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[44] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: A High-availability and Integrity Layer for Cloud Storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 187–198, New York, NY, USA, 2009. ACM.

[45] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live Wide-area Migration of Virtual Machines Including Lo-

cal Persistent State. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07)*, pages 169–179, New York, NY, USA, 2007. ACM.

[46] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: Cloud Micro-elasticity via VM State Coloring. In *Proceedings of the Sixth European Conference on Computer Systems (EuroSys'11)*, pages 273–286, New York, NY, USA, 2011. ACM.

[47] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and Software Support for Virtualization*. Morgan & Claypool Publishers, 2017.

[48] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of the 11th International Conference on Quality of Service (IWQoS'03)*, pages 381–398, Berlin, Heidelberg, 2003. Springer-Verlag.

[49] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *NSDI*, volume 8, pages 337–350, 2008.

[50] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[51] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI'05)*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[52] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM J. Res. Dev.*, 25(5):483–490, September 1981.

[53] Scott W. Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization System Including a Virtual Machine Monitor for a Computer with a Segmented Architecture, Feb 2002.

[54] Jeff Dike. A user-mode port of the Linux kernel. In *Annual Linux Showcase & Conference*, 2000.

[55] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: Robust Hybrid Cloud Storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*, pages 12:1–12:14, New York, NY, USA, 2014. ACM.

[56] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS'03)*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[57] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[58] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 137–151, New York, NY, USA, 1996. ACM.

[59] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. Frenetic: A high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[60] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, HOTOS'07, pages 6:1–6:6, Berkeley, CA, USA, 2007. USENIX Association.

[61] Sharath George. Usermode kernel: running the kernel in userspace in VM environments. Master's thesis, University of British Columbia, Dec 2008.

[62] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 475–490, Bellevue, WA, 2012. USENIX.

[63] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload Analysis and Demand Prediction of Enterprise Data Center Ap-

plications. In *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization (IISWC '07)*, pages 171–180, Washington, DC, USA, 2007. IEEE Computer Society.

[64] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the June 4-8, 1973, national computer conference and exposition (AFIPS'73)*, pages 309–318, New York, NY, USA, 1973. ACM.

[65] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[66] Alexander Graf and Joerg Roedel. Nesting the virtualized world. In *Linux Plumbers Conference*, 2009.

[67] Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. Seagull: Intelligent Cloud Bursting for Enterprise Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 33–33, Berkeley, CA, USA, 2012. USENIX Association.

[68] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[69] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM.

[70] Qing He. Nested virtualization on Xen. *Xen Summit Asia*, 2009.

[71] Jason Hennessey, Sahil Tikale, Ata Turk, Emine Ugur Kaynar, Chris Hill, Peter Desnoyers, and Orran Krieger. Hil: Designing an exokernel for the data center. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 155–168, New York, NY, USA, 2016. ACM.

[72] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: A Consolidation Manager for Clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual*

Execution Environments (VEE'09), pages 41–50, New York, NY, USA, 2009. ACM.

[73] Michael R. Hines and Kartik Gopalan. Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-paging and Dynamic Self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[74] Takahiro Hirofuchi, Hidemoto Nakada, Hirotaka Ogawa, Satoshi Itoh, and Satoshi Sekiguchi. A Live Storage Migration Mechanism over Wan and Its Performance Evaluation. In *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '09, pages 67–74, New York, NY, USA, 2009. ACM.

[75] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 341–354, New York, NY, USA, 2007. ACM.

[76] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Supercloud: Opportunities and challenges. *SIGOPS Oper. Syst. Rev.*, 49(1):137–141, January 2015.

[77] Flavio Paiva Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256. IEEE, 2011.

[78] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM.

[79] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC'09)*, pages 117–126, New York, NY, USA, 2009. ACM.

[80] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[81] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*, pages 101–110, New York, NY, USA, 2009. ACM.

[82] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[83] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. SafeStore: A Durable and Practical Storage System. In *Proceedings of the USENIX Annual Technical Conference (ATC'07)*, pages 10:1–10:14, Berkeley, CA, USA, 2007. USENIX Association.

[84] Lagar-Cavilla, Horacio Andrés and Whitney, Joseph Andrew and Scannell, Adin Matthew and Patchin, Philip and Rumble, Stephen M. and de Lara, Eyal and Brudno, Michael and Satyanarayanan, Mahadev. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*, pages 1–12, New York, NY, USA, 2009. ACM.

[85] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 33–48, New York, NY, USA, 1983. ACM.

[86] Hugh C. Lauer and David Wyeth. A recursive virtual machine architecture. In *Proceedings of the workshop on virtual computer systems*, pages 113–116, New York, NY, USA, 1973. ACM.

[87] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun.*, 14(7):1280–1297, September 2006.

[88] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated Control for Elastic Storage. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*, pages 1–10, New York, NY, USA, 2010. ACM.

[89] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[90] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. The Design and Evolution of Live Storage Migration in VMware ESX. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 14–14, Berkeley, CA, USA, 2011. USENIX Association.

[91] E. Michael Maximilien, Ajith Ranabahu, Roy Engehausen, and Laura Anderson. IBM Altocumulus: a cross-cloud middleware and platform. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*, pages 805–806, New York, NY, USA, 2009. ACM.

[92] G. McGrath and P. R. Brenner. Serverless Computing: Design, Implementation, and Performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, June 2017.

[93] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[94] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[95] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[96] Bogdan Nicolae and Franck Cappello. A Hybrid Local Storage Transfer Scheme for Live Migration of I/O Intensive Workloads. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 85–96, New York, NY, USA, 2012. ACM.

[97] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 interpretive-

execution architecture, foundation for VM/ESA. *IBM Systems Journal*, 30(1):34–51, 1991.

[98] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*, pages 13–26, New York, NY, USA, 2009. ACM.

[99] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[100] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.

[101] Daniel Price and Andrew Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 241–254, Berkeley, CA, USA, 2004. USENIX Association.

[102] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *USENIX Security Symposium*, 2003.

[103] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333, June 2010.

[104] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[105] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A framework for building per-application library operating systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 671–688, GA, 2016. USENIX Association.

[106] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 16:1–16:15, New York, NY, USA, 2015. ACM.

[107] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[108] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, Jakub Szefer, and Hakim Weatherspoon. Towards weakly consistent local storage systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 294–306, New York, NY, USA, 2016. ACM.

[109] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, Berkeley, CA, USA, 2012. USENIX Association.

[110] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM.

[111] Weijia Song, Zhen Xiao, Qi Chen, and Haipeng Luo. Adaptive Resource Provisioning for the Cloud Using Online Bin Packing. *IEEE Transactions on Computers*, 63(11):2647–2660, Nov 2014.

[112] Christopher Stewart, Terence Kelly, Alex Zhang, and Kai Shen. A Dollar from 15 Cents: Cross-platform Management for Internet Services. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08)*, pages 199–212, Berkeley, CA, USA, 2008. USENIX Association.

[113] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[114] Chunqiang Tang. FVD: A High-performance Virtual Machine Image Format for Cloud. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2011. USENIX Association.

[115] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela

Oliveira, and Donald E. Porter. Cooperation and security isolation of Library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[116] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[117] Anthony Velte and Toby Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[118] P. Verissimo, A. Bessani, and M. Pasin. The TClouds architecture: Open and resilient cloud-of-clouds computing. In *Dependable Systems and Networks Workshops (DSN-W), IEEE/IFIP 42nd International Conference on*, June 2012.

[119] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 181–194, New York, NY, USA, 2002. ACM.

[120] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 3–14, New York, NY, USA, 2010. ACM.

[121] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[122] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The Xen-Blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, pages 113–126, 2012.

[123] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus van der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proceedings of the 7th ACM SIG-*

PLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11), pages 121–132, New York, NY, USA, 2011. ACM.

[124] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)*, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

[125] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[126] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 292–308, New York, NY, USA, 2013. ACM.

[127] Zhen Xiao, Weijia Song, and Qi Chen. Dynamic Resource Allocation Using Virtual Machines for Cloud Computing Environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117, June 2013.

[128] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM.

[129] Jie Zheng, Tze Sing Eugene Ng, and Kunwadee Sripanidkulchai. Workload-aware Live Storage Migration for Clouds. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 133–144, New York, NY, USA, 2011. ACM.

[130] Xiaoyun Zhu, Don Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Rob Gardner, Tom Christian, and Lucy Cherkasova. 1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center. In *Proceedings of the 2008 International Conference on Autonomic Computing (ICAC'08)*, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.